Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume

As Published: https://doi.org/10.1007/s00446-019-00363-1

Publisher: Springer Berlin Heidelberg

Persistent URL: https://hdl.handle.net/1721.1/131299

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume: POWER8 TM

Cite this article as: Shady Issa, Pascal Felber, Alexander Matveev and Paolo Romano, Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume: POWER8 TM, Distributed Computing https://doi.org/10.1007/s00446-019-00363-1

Distributed Computing manuscript No.
(will be inserted by the editor)

Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume

POWER8 TM

Shady Issa · Pascal Felber · Alexander Matveev · Paolo Romano

Received: date / Accepted: date

Abstract Transactional memory (TM) aims at simplifying concurrent programming via the familiar abstraction of atomic transactions. Recently, Intel and IBM have integrated hardware based TM (HTM) implementations in commodity processors, paving the way for the mainstream adoption of the TM paradigm. Yet, existing HTM implementations suffer from a crucial limitation, which hampers the adoption of HTM as a general technique for regulating concurrent access to shared memory: the inability to execute transactions whose working sets exceed the capacity of CPU caches. In this article we propose P8TM, a novel approach that mitigates this limitation on IBM's POWER8 architecture by leveraging a key combination of hardware and software techniques to support different execution paths. P8TM also relies on self-tuning mechanisms aimed at dynamically switching between different execution modes to best adapt to the workload characteristics. In-depth evaluation with several benchmarks indicates that P8TM can achieve striking performance gains in workloads that stress the capacity limitations of HTM, while achieving performance on par with HTM even in unfavourable workloads.

Keywords Hardware Transactional Memory · Parallel Programming · Concurrency · Self-tuning

Shady Issa
INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal
E-mail: shadi.issa@tecnico.ulisboa.com

Pascal Felber
University of Neuchâtel, Switzerland
E-mail: pascal.felber@unine.ch

Alexander Matveev
MIT, USA
E-mail: amatveev@csail.mit.edu

Paolo Romano
INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal
E-mail: romano@inesc-id.pt

1 Introduction

Transactional memory (TM) has emerged as a promising paradigm that aims at simplifying concurrent programming by bringing the familiar abstraction of atomic and isolated transactions to the domain of parallel computing. Unlike when using locks to synchronize access to shared data or code portions, with TM programmers need only specify *what* is synchronized and not *how* synchronization should be performed. This results in simpler designs that are easier to write, reason about, maintain, and compose [40].

Over the last years, the relevance of TM has been growing along with the maturity of available implementations for this new paradigm, both in terms of integration at the programming language as well as at the architectural level. On the front of integration with programming languages, a recent milestone has been the official integration of TM in mainstream languages, such as C/C++ [5]. On the architecture's side, the appearance of hardware implementations of the TM abstraction (HTM) in Intel's [46] and IBM's [30, 44] processors represented another major breakthrough.

Commercially available hardware implementations share various architectural choices, although they do come in different flavours [26, 30, 46]. The key common trait of current HTM systems is their best effort nature: current implementations maintain transactional metadata (e.g., memory addresses read/written by a transaction) in the processor's cache and rely on relatively

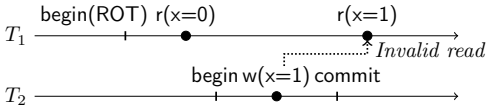

Fig. 1 ROTs do not track reads and may, as such, observe different values when reading the same variable multiple times.

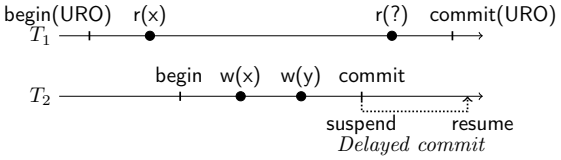

Fig. 2 In order to preserve consistency, the write back of shared variables updated by an update transaction must be delayed until after any URO transaction has completed execution.

non-intrusive modification to the pre-existing cache coherency protocol to detect conflict among concurrent transactions. Due to the inherently limited nature of processor caches, current HTM implementations impose stringent limitations on the number of memory accesses that can be performed within a transaction,[1] hence providing no progress guarantee even for transactions that run in absence of concurrency. As such, HTM requires a fallback synchronization mechanism (also called *fallback path*), which is typically implemented via a pessimistic scheme based on a single global lock.

Despite these common grounds, current HTM implementations have also several relevant differences. Besides internal architectural choices (e.g., where and how in the cache hierarchy transactional metadata are maintained), Intel's and IBM's implementations differ notably by the programming interfaces they expose. In particular, IBM POWER8's HTM implementation extends the conventional transactional demarcation API (to start, commit and abort transactions) with two additional, unique features [6]:

- *Suspend/resume*: the ability to suspend and resume a transaction, allowing, between the suspend and resume calls, for the execution of instructions/memory accesses that escape from the transactional context.
- *Rollback-only transaction (ROT)*: a lightweight form of transaction that has lower overhead than regular transactions but also weaker semantics. In particular ROTs avoid tracking in hardware the addresses accessed by load operations. As such, ROTs cannot detect write-after-read conflicts and do not guarantee isolation, as illustrated in Figure 1. However, they still ensure the atomicity of the stores issued by a transaction, which appear to be all executed as a unit or not executed at all.

In this work we present POWER8 TM (P8TM), a novel TM that exploits these two specific features of POWER8's HTM implementation in order to overcome (or at least mitigate) what is, arguably, the key limitation stemming from the best-effort nature of existing HTM systems: the inability to execute transactions whose working sets exceed the capacity of CPU caches. P8TM pursues this objective via an innovative hardware-software co-design that leverages several novel techniques, which we overview in the following.

[1] The list of restrictions is actually longer, including the lack of support for system calls and other non-undoable instructions, context switches and ring transitions.

Uninstrumented read-only transactions (UROs). P8TM executes read-only transactions outside of the scope of hardware transactions, hence sparing them from the spurious aborts and capacity limitations that affect HTM, while still allowing them to execute concurrently with update transactions , as depicted in Figure 2. This result is achieved by exploiting POWER8's suspend/resume mechanism to implement a quiescence scheme, similar in spirit to Read-Copy Update (RCU) [23, 24, 33], which shelters UROs from observing inconsistent snapshots that reflect the commit events of concurrent update transactions.

ROT-based update transactions. In typical TM workloads the read/write ratio tends to follow the 80/20 rule, i.e., transactified methods tend to have large read-sets and much smaller write sets [14]. This observation led us to develop a novel concurrency control scheme based on a novel hardware-software co-design. It combines the hardware-based ROT abstraction—which tracks only transactions' write sets, but not their read-sets, and, as such, does not guarantee isolation—with software based techniques aimed to preserve correctness in the presence of concurrently executing ROTs, UROs, and plain HTM transactions. Specifically, P8TM relies on a novel mechanism, which we call Touch-To-Validate (T2V), to execute concurrent ROTs safely. T2V relies on a lightweight software instrumentation of reads within ROTs' and a hardware aided validation mechanism of the read-set during the commit phase.

HTM-friendly (software-based) read-set tracking. A key challenge that we had to tackle while implementing P8TM is to develop a "HTM-friendly" software-based read-set tracking, i.e., designed to be highly efficient when employed in the context of a hardware transaction. In fact, all the memory writes issued from within a ROT are transparently and automatically tracked by the hardware, including the writes issued by the software-based

read-set tracking mechanism for ROTs. Thus, the read-set tracking mechanism can consume cache capacity that could be otherwise used to accommodate application-level writes issued from within a ROT. P8TM tackles this issue by integrating two read-set tracking mechanisms that explore different trade-offs between space and time efficiency.

Self-tuning. In order to ensure robust performance in a broad range of workloads, P8TM integrates a lightweight reinforcement learning mechanism (based on the UCB algorithm [29]) that automates the decision of whether to: *(i)* use upfront ROTs and UROs, avoiding using the HTM at all; *(ii)* first attempt transactions in HTM, and then fallback to ROTs/UROs in case of capacity exceptions; or *(iii)* completely switch off ROTs/UROs, using only HTM. The use of self-tuning allows P8TM to identify the scheduling policy that best fits the workload characteristics in a fully transparent way to programmers.

We evaluated P8TM by means of extensive study that encompasses synthetic micro-benchmarks, the benchmarks in the STAMP suite [35], as well as a porting to TM of the popular TPC-C benchmark [43]. The results of our study show that P8TM can achieve up to ∼7× throughput gains with respect to plain HTM and extend its capacity by more than one order of magnitude, while remaining competitive even in unfavorable workloads.

2 Related Work

The quiescence mechanism employed by P8TM to protect UROs from witnessing inconsistent states is similar in spirit to Read-Copy-Update (RCU) [33]. RCU is a synchronization mechanism which eliminates the need for acquiring locks when accessing shared data without modifying it. With RCU, readers, or threads that access shared data without modifying it, only need to advertise their status atomically, before and after the read-only critical section. To shelter readers from inconsistent states, a writer, or a thread that would modify shared data, creates a copy of the data to be modified and apply its updates to the copy. The new copy must be applied in a way such that readers that existed before the writer continue to access the old data, while new readers get to witness the updated state. The old data can only be discarded when it is safe to do so, i.e., when all the readers that existed prior to the writer have finished their critical section . Compared to P8TM, RCU has two major limitations: it allows only a single writer at a time and can only support data structures where switching between old and updated versions can be performed atomically. The latter is arguably the reason behind the small number of RCU-based data structures [2,8,34]. We compare P8TM against RCU in our micro-benchmarks and we show that P8TM does yield higher throughput in both read and write-dominated workloads.

Since the introduction of HTM support in mainstream commercial processors by Intel and IBM, several experimental studies have aimed to characterize their performance and limitations [17, 21, 36]. An important conclusion reached by these studies is that HTM's performance excels with workloads that fit the hardware capacity limitations. Unfortunately, though, HTM's performance and scalability can be severely hampered in workloads that contain even a small percentage of transactions that do exceed the hardware's capacity. This is due to the need to execute such transactions using a sequential fallback mechanism based on a single global lock (SGL), which causes the immediate abort of any concurrent hardware transactions and prevents any form of parallelism.

Hybrid TM [10, 28] (HyTM) attempts to address this issue by falling back to software-based TM (STM) implementations when transactions cannot successfully execute in hardware. Hybrid NOrec (HyNOrec) is probably one of the most popular and effective HyTM designs proposed in the literature. HyNOrec [9] falls back on using the NOrec STM, which lends itself naturally to serve as fallback for HTM. In fact, NOrec uses a single versioned lock for synchronizing (software) transactions. Synchronization between HTM and STM can hence be attained easily, by having HTM transactions update the versioned lock used by NOrec. Unfortunately, the coupling via the versioned lock introduces additional overheads on both the HTM and STM side, and can induce spurious aborts of HTM transactions. Further, HyNOrec prohibits concurrency between HTM and committing STM, even in the absence of conflict.

Recently, RHyNOrec [32] proposed to decompose a transaction running on the fallback path into multiple hardware transactions: a read-only prefix and a single post-fix that encompasses all the transaction's writes, with regular NOrec shared operations in between. This can reduce the false aborts that would otherwise affect hardware transactions in HyNOrec. Unfortunately, though, this approach is only viable if the transaction's post-fix, which may potentially encompass a large number of reads, does fit in hardware. Further, the technique used to enforce atomicity between the read-only and the remaining reads relies on fully instrumenting every read within the prefix hardware transaction. This utterly limits the capacity—and consequently the practicality—of these transactions. Unlike RHyNOrec, P8TM can execute read-only transactions of arbitrary length in a fully

uninstrumented way. Further, the T2V mechanism employed by P8TM to validate update transactions relies on a much lighter and efficient read-set tracking and validation schemes that can even further increase the capacity of transactions.

Our work is also related to the literature aimed to enhance HTM's performance by optimizing the management of the SGL fallback path. A simple, yet effective optimization, which we include in P8TM, is to avoid the so called *lemming effect* [13] by ensuring that the SGL is free before starting a hardware transaction. An alternative solution to the same problem is the use of an auxiliary lock [1]. In our experience, these two solutions provide equivalent performance, so we opted to integrate in P8TM the former, simpler, approach. Calciu et al. [7] suggested lazy subscription of the SGL in order to decrease the vulnerability window of HTM transactions. However, this approach was shown to be unsafe in subtle scenarios that are hard to fix using automatic compiler-based techniques [12].

P8TM integrates a self-tuning approach that shares a common theoretical framework (the UCB reinforcement learning algorithm [29]) with Tuner [15]. However, Tuner addresses an orthogonal self-tuning problem to the one we tackled in P8TM: Tuner exploits UCB to identify the optimal retry policy before falling back to the SGL path upon a capacity exception; in P8TM, conversely, UCB is to determine which synchronization to use (e.g., ROTs/UROs vs. plain HTM). Another recent work that makes extensive use of self-tuning techniques to optimize HTM's performance is SEER [16]. Just like Tuner, SEER addresses an orthogonal problem—defining a scheduling policy that seeks an optimal trade-off between throughput and contention probability—and could, indeed, be combined with P8TM.

Finally, P8TM builds on and extends on HERWL [19], where we introduced the idea of using POWER8's suspend-resume and ROT facilities to elide read-write locks. Besides targeting a different application domain (transactional programs vs. lock elision), P8TM integrates a set of novel techniques. Unlike HERWL, P8TM supports the concurrent execution of update transactions in ROTs. Achieving this result implied introducing a novel concurrency control mechanism (which we named Touch-To-Validate). Additionally, P8TM integrates self-tuning techniques that ensure robust performance also in unfavorable workloads.

3 Background on HTM

From the software perspective, Hardware Transactional Memory (HTM) extends a processor's instruction set with new instructions (typically, *begin*, *commit* and *abort*) that allows for demarcating blocks of code as transactions. The hardware, then, guarantees strong atomicity of these transactions, i.e., all the operations executed within a transaction (transactional accesses) would either appear executed or not at all to any other concurrent code (whether transactional or non-transactional). Current HTM implementations achieve this by extending the cache coherency protocol to detect conflicts between code executed from within a transaction and any concurrent code. Accesses from within a transaction are tracked, either in the CPU caches or dedicated buffers. A conflict is detected when one of those tracked accesses overlap with another transactional or non-transactional access that took place after the tracked access and before the transaction has successfully committed, while at least one of the accesses is a store. When a conflict is detected, the hardware automatically triggers the abort of the transaction (in case of a conflict between two transactions, at least one of them is guaranteed to be aborted). The available HTM systems detect conflicts at the granularity of a single cache line [36], i.e., accesses overlap when they reference bytes that lie on the same cache line. Throughout the lifetime of a transaction, updates are buffered and applied only upon successful commit of the transaction during a write-back phase, which guarantees that these updates appear atomically. In case of an abort, the buffered updates are discarded without any side effects.

HTM transactions may still abort even in the absence of conflicts due to several reasons such as timer interrupts, executing non-supported operations, page faults and exceeding the capacity of the hardware resources dedicated for tracking transactional accesses. The latter is arguably one of the main limitations that hinder the practical adoption of HTM as a general purpose synchronization mechanism [17, 21]. In this paper we are proposing P8TM to tackle this limitation in the POWER8 HTM. The transactional capacity (64 cache lines) in POWER8 is bounded by an 8KB cache, called TMCAM, which stores the addresses of the cache lines read or written within the transaction.

When programs request to start a transaction, a *started* code is placed in the, so called, status buffer by the hardware. If, later, the transaction aborts, the hardware stores in the status buffer a code describing the cause of the abort and alters the program counter to jump back to the instruction immediately following the transaction begin. Hence, in order to distinguish whether a transaction has just started, or has undergone an abort, programs must test the status code returned after beginning the transaction.

As mentioned, in addition to HTM transactions, POWER8 also supports Rollback-Only Transactions (ROT). ROTs are a special type of transactions supported by POWER8 that are meant to provide single thread failure-atomicity, i.e., the ability to roll-back the execution of a code block. The main difference being that in ROTs, only the writes are tracked in the TMCAM, giving virtually infinite read-set capacity. Reads performed by ROTs are essentially treated as non-transactional reads. As a consequence, ROTs have a larger capacity than normal transactions, but they do not guarantee safety in presence of concurrent executions .In fact, since with ROTs reads are not tracked, it is not possible to detect write-after-read conflicts, i.e., conflicts that manifest when a location that is accessed in read mode by a ROT is then updated (by a different thread) before the ROT commits.

In POWER8, both HTM transactions and ROTs detect conflict eagerly, i.e., they are aborted as soon as they incur a conflict. The only exception is when they incur a conflict while in suspend mode: in this case, they abort only once they resume. Finally, P8TM exploits how POWER8 manages conflicts that arise between non-transactional code and HTM transactions/ROTs, i.e., if a HTM transaction/ROT issues a write on X and, before it commits, a non-transactional read/write is issued on X, the HTM transaction/ROT is immediately aborted by the hardware.

4 P8TM Overview

The key challenge in designing execution paths that can run concurrently with HTM is efficiency: it is hard to provide a software-based path that executes concurrently with the HTM path, while preserving correctness and speed. The main problem is that the protocol must make the hardware aware of concurrent software memory reads and writes, which requires to introduce expensive tracking mechanisms in the HTM path.

P8TM tackles this issue by exploiting two unique features of the IBM POWER8 architecture: (1) suspend/resume for hardware transactions, and (2) ROTs. P8TM combines these new hardware features with a RCU-like quiescence scheme in a way that avoids the need to track reads in hardware. This can in particular reduce the likelihood of capacity aborts that would otherwise affect transactions that perform a large number of reads.

The key idea is to provide two novel execution paths alongside the HTM path: *(i)* a *URO path*, which executes read-only transactions without any instrumentation, and *(ii)* a *ROT path*, which executes update transactions that do not fit in HTM as ROTs.

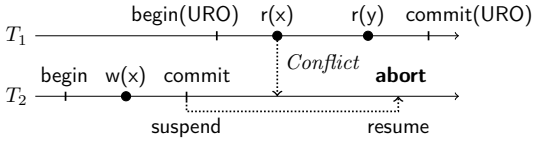

Fig. 3 A read access to a shared variable updated by a suspended update transaction will abort the latter (when it resumes).

HTM transactions and ROTs exploit the speculative hardware support to hide writes from concurrent reads. This allows coping with read-write conflicts that occur during ROTs/UROs, but it does not cover writer-after-read conflicts that occur after the commit of an update transaction. For this purpose, before an update transaction (running either as a HTM transaction or a ROT) commits, it first suspends itself and then executes a quiescence mechanism that waits for the completion of currently executing ROTs/UROs. In addition to that, in case the committing update transaction is enclosed in a ROT, it further executes an original *touch-based validation* step, which is described later, before resuming and committing. This process of "suspending and waiting" ensures that the writes of an update transaction will be committed only if they do not target/overwrite any memory location that was previously read by any concurrent ROT/URO.

4.1 Uninstrumented Read-Only Transactions

P8TM exploits the suspend/resume mechanism to execute read-only transactions without resorting to the use of hardware transactions or performing instrumentation of read operations on shared data (URO path). This provides the key benefit of ensuring strong progress guarantees for read-only transactions, which are spared by spurious (and repeated) aborts caused by the underlying HTM implementation.

Let us assume, for simplicity, that update transactions execute only using HTM (the case of ROT-based update transactions is analogous and will be discussed in more detail in Section 5.3). HTM transactions (and ROTs) buffer memory writes until the point of commit, hence, concurrent read-only transactions can safely execute with update transactions, as long as the latter ones do not commit. In fact, any read performed by a URO after a conflicting write of a concurrent update transaction will immediately abort the latter.

However, it is unsafe for update transactions to commit when there are concurrent UROs. This is illustrated in Figure 2 where an uninstrumented read-only transaction (T_1) and an update transaction (T_2) concurrently

access two shared variables. As T_2 fully executes between two read accesses by T_1, T_2 cannot detect the concurrent execution of T_1 and, by committing, T_2 would expose T_1 to an inconsistent snapshot that may contain a mix of old and new values (if $r(?) \neq r(y)$ in the figure). To overcome this problem, a key idea in P8TM is to suspend the hardware speculation of an update transaction, and then wait for all current UROs to complete by using an RCU-like (epoch-based) quiescence mechanism [23, 24, 33]. This suspend-wait sequence has a two-fold effect. First, it drains all current read-only transactions that may read a location written by a suspended update transaction, as these may be exposed to inconsistent snapshots if the update transaction committed before their completion (as illustrated in Figure 2). Second, any read issued to a location previously written by a suspend hardware transaction will cause the abort of the latter, as illustrated in Figure 3. As a result, after the wait is complete, it is safe to commit the update transaction, so P8TM simply resumes hardware speculation and issues a commit request.

4.2 Touch-based Validation

Touch-To-Validate (T2V) is another core mechanism of P8TM that enables safe and concurrent execution of ROT-based update transactions. As already mentioned, in fact, ROTs do not track read accesses in hardware. As such, their concurrent execution is generally unsafe, as illustrated by the example in Figure 1. Thread T_1 starts a ROT and reads x. At this time, thread T_2 starts a concurrent ROT, writes a new value to x, and commits. As ROTs do not track reads, they are unable to detect write-after-read conflicts. As such, the ROT of T_1 does not get aborted and can read inconsistent values (e.g., the new value of x). To avoid such scenarios T2V leverages two key mechanisms that couple: *(i)* software-based tracking of read accesses; and *(ii)* hardware- and software-based read-set validation during the commit phase.

For the sake of clarity, assume that threads only execute ROTs — we will consider other execution modes later. A thread can be in one of three states: *inactive*, *active*, and *committing*. A thread that executes non-transactional code is inactive. When the thread starts a ROT, it enters the active phase and starts tracking, in software, each read access to shared variables by logging the associated memory address in a special data structure called *rot-rset*. Finally, when the thread finishes executing its transaction, it enters the committing phase. At this point, it has to wait for concurrent threads that are in the active phase to either enter the commit phase or become inactive (upon abort). Thereafter, the committing thread traverses its *rot-rset* and re-reads each address before eventually committing.

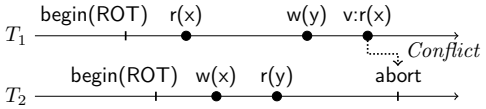

Fig. 4 By re-reading x during *rot-rset* validation at commit time (denoted by v:r), T_1 forces the abort of T_2 that has updated x in the meantime.

The goal of this validation step is to "touch" each previously read memory location in order to abort any concurrent ROT that might have written to the same address. For example, in Figure 4, T_1 re-reads x during *rot-rset* validation. At that time, T_2 has concurrently updated x but has not yet committed, and it will therefore abort (remember that ROTs track and detect conflicts for writes). This allows T_1 to proceed without breaking consistency: indeed, ROTs buffer their updates until commit and hence the new value of x written by T_2 is not visible to T_1. Note that adding a simple quiescence phase before commit, without performing the *rot-rset* validation, cannot solve the problem in this scenario.

The originality of the T2V mechanism is that the ROT does not use read-set validation for verifying that its read-set is consistent, as many STM algorithms do, but to trigger hardware conflicts detection mechanisms. This also means that the values read during *rot-rset* validation are irrelevant and ignored by the algorithm.

5 P8TM Algorithm

This section provides a detailed description of P8TM's algorithm. For the sake of clarity, we present P8TM in an incremental fashion. We start by describing the management of the URO path (Section 5.1) and of the ROT path (Section 5.2), each on its own. Then, in Section 5.3 we provide a complete description of the algorithm, by discussing *(i)* how to extend the ROT path to first attempt using HTM transactions, and *(ii)* how to synchronize the URO and ROT paths with the pessimistic fallback path (based on a single global lock).

Finally, we discuss the correctness, fairness and progress guarantees of our proposed solution in Sections 5.4 and 5.5, respectively.

5.1 URO Path

Let us start by considering an initial version of the P8TM algorithm (Algorithm 1) that assumes that read-only transactions execute in the URO path, and that

Algorithm 1 P8TM: URO path only algorithm

```
1:  Shared variables:
2:      status[N] ← {⊥, ⊥, ..., ⊥}        ▷ One per thread
3:  Local variables:
4:      tid ∈ [0..N]                       ▷ Identifier of current thread
5:  function SYNCHRONIZE
6:      s[N] ← status                      ▷ Read all statuses
7:      for i ← 0 to N−1 do                ▷ Wait until all threads...
8:          if s[i] is ACTIVE then         ▷ ...running UROs...
9:              wait until status[N] ≠ s[i]   ▷ ...end
10: function BEGIN_RO
11:     status[tid] ← ACTIVE               ▷ Update thread's status
12:     MEM_FENCE                          ▷ Ensure visibility to update txs.
13: function COMMIT_RO
14:     MEM_FENCE                          ▷ Avoid re-ordering.
15:     status[tid] ← ⊥                    ▷ Reset thread's status
16: function BEGIN_W                       ▷ Start update tx.
17:     repeat until TX_BEGIN = STARTED
18: function COMMIT_W
19:     tx_suspend                         ▷ Suspend transaction
20:     SYNCHRONIZE()                      ▷ Let UROs drain their reads
21:     tx_resume                          ▷ Resume transaction
22:     tx_commit                          ▷ Write back updates
```

update transactions execute using plain HTM transactions. For simplicity, this version of the algorithm blindly retries failed update transactions, irrespective of the abort cause.

To ensure proper synchronization with update transactions, P8TM must keep track of which UROs are executing. This is achieved by having every thread maintain a status variable that is set and unset in the BEGIN_RO() and COMMIT_RO() functions when respectively starting and ending a read-only transaction.

Update transactions are started and committed by calling the BEGIN_W() and COMMIT_W() functions. They execute as plain HTM transactions, hence, throughout the execution of an update transaction, the memory writes are buffered and, thus, hidden from UROs.

Assume there is an update transaction and a concurrent URO that, respectively, update and read the same shared variable. If the memory access in the URO path occurs *after* the update transaction has written the variable, then the update transaction will immediately abort and restart. If however the read occurs *before* the update transaction issues the write access, then no conflict will be detected and the URO will be serialized before the update transaction.

When an update transactions completes its execution, it must issue a commit request in order to write back its (speculative) updates. Yet, doing so without precaution would break consistency, since a URO might see a mix of old and new data (prior and after the commit of the update transaction).

Therefore, before committing, an update transaction waits for all UROs that *might* have read any of the locations it has written to. Since P8TM does not keep track of which memory locations have been accessed by UROs (which would require software instrumentation of memory accesses), it relies on a lightweight, RCU-like, quiescence mechanism that waits for the completion of any URO found active at the beginning of the quiescence phase. This is implemented in the SYNCHRONIZE() function by reading the status of each thread once and waiting for all *active* to change value. Note that this quiescence mechanism does not prevent the start of new read-only transactions, nor forces a suspended update transaction to wait for read-only transactions activated after the start of its quiescence phase. This is safe, since read-after-write conflicts will be handled as described above, i.e., by aborting the update transaction.

An additional challenge is that the quiescence barrier cannot be implemented straightforwardly in the context of hardware transaction. The problem is that if a URO updates its status that is being monitored by some concurrent update transaction, this will be detected as a write-after-read conflict, and lead to the abort of the update transaction.

To tackle this issue, P8TM exploits the suspend/resume feature of the POWER8 microarchitecture, which allows to temporarily suspend the active transaction, perform non-transactional operations, and later resume the transaction. P8TM relies on this feature to execute the quiescence phase and allow update transactions to monitor the status of concurrent UROs without incurring spurious aborts.

Note that any conflict occurring while a transaction is suspended will trigger an abort upon its resume, hence protecting concurrent UROs from seeing inconsistent snapshots. Indeed, consider a URO that starts after calling SYNCHRONIZE(), i.e., which has not been found active by an update transaction upon the start of its quiescence phase. This URO will execute concurrently with the write-back phase of the update transaction. If the URO reads any memory location that has been updated by the update transaction before this completes its write-back phase (which is atomic), then the latter will abort; else, if the read is issued after the completion of the write-back phase, the URO will see the new version.

5.2 ROT Path

We now present a version of the P8TM algorithm (Algorithm 2) assuming only the existence of update transactions running in the ROT path. Also in this case, for simplicity, we blindly retry to execute failed ROTs irrespective of the abort cause.

Algorithm 2 P8TM: ROT path only algorithm

```
1:  Shared variables:
2:      status[N] ← {⊥, ⊥, ..., ⊥}        ▷ One per thread
3:  Local variables:
4:      tid ∈ [0..N]                       ▷ Identifier of current thread
5:      rot-rset ← ∅                       ▷ Transaction's read-set
6:  function BEGIN_ROT
7:      repeat                             ▷ Blindly retry ROT
8:          status[tid] ← ACTIVE           ▷ Update status
9:          MEM_FENCE                      ▷ Make sure others know
10:         rot-rset ← ∅                   ▷ Clear read-set
11:         tx ← tx_begin_rot              ▷ Start ROT
12:     until tx = STARTED                 ▷ Repeat until success...
13: function READ(addr)                    ▷ Read shared variable
                                           ▷ Track ROT reads
14:     rot-rset ← rot-rset ∪{addr}
15: function SYNCHRONIZE
                          ▷ Read and copy all status variables
16:     s[N] ← status
17:     for i ← 0 to N−1 do                ▷ Wait until all threads...
18:         if s[i] = ACTIVE then          ▷ ...that are active...
19:             wait until status[i] ≠ s[i]    ▷ ...end
20: function TOUCH_VALIDATE
21:     for addr ∈ rot-rset do             ▷ Re-read all elements...
22:         read *addr                     ▷ ...from read-set
23: function COMMIT_ROT
24:     tx_suspend                         ▷ Suspend ROT
25:     status[tid] ← ROT-COMMITTING       ▷ Tell others...
26:     MEM_FENCE                          ▷ ...we are committing
27:     tx_resume                          ▷ Resume ROT
28:     SYNCHRONIZE()                      ▷ Quiescence inside ROT
29:     TOUCH_VALIDATE()                   ▷ Touch to validate
30:     tx_commit_rot                      ▷ Commit ROT
31:     status[tid] ← ⊥
```

To start an update transaction, a thread first lets others know that it is *active* and initializes its data structures before actually starting a ROT (Lines 8–11). Then, during ROT execution, it just keeps track of reads to shared data by adding them to the thread-local *rot-rset* (Line 14). To complete the ROT, the thread first announces that it is *committing* by setting its shared *status* variable. Note that this is performed while the ROT is suspended (Lines 24–27) because otherwise the write would be buffered and invisible to other threads.

Next, the algorithm quiesces by waiting for all threads that are in a ROT to at least reach their commit phase (Lines 15–19). It then executes the touch-based validation mechanism, which simply consists in re-reading all address in the *rot-rset* (Lines 20–22), before finally committing the ROT (Line 30) and resetting the *status*.

5.3 Complete Algorithm

The naive approach of the basic algorithm to only use ROTs is unfortunately not practical nor efficient in real-world settings for two main reasons: (1) ROTs only provide "best effort" properties and thus a fallback is needed to guarantee liveness; and (2) using ROTs for short transactions that fit in a regular HTM transaction is inefficient because of the overhead of the software-based read tracking and validation mechanisms. Therefore, we extend the algorithm so that if first tries to use regular hardware transactions, then upon failure switches to ROTs, and finally falls back to a global lock (GL) in order to guarantee progress. Note that when executing using plain HTM or in the GL, threads do not need to set their *status* variable. The pseudo-code of the complete algorithm is shown in Algorithms 3 and 4.

For HTM transactions and ROTs to execute concurrently, the former must delay their commit until completion of all active ROTs. This is implemented using an RCU-like quiescence mechanism as in the URO algorithm (Lines 69–73). Note that a simple quiescence, without a validation step afterwards, is sufficient in this case.

In this version, we do not blindly retry aborted hardware transactions and ROTs. Conversely, we exploit the status code returned by TX_BEGIN/TX_BEGIN_ROT to determine which retry policy to use. If the return code of TX_BEGIN/TX_BEGIN_ROT is `STARTED`, indicating success, the HTM transaction/ROT can start executing speculatively. If an abort happens during execution of a HTM transaction/ROT, then control jumps back to just after the call to TX_BEGIN/TX_BEGIN_ROT and the status code contains information about the failure cause. For the sake of simplicity, we assume that the status code can be `STARTED`, `TRANSIENT-ABORT`, or `CAPACITY-ABORT` to respectively indicate if the transaction executes speculatively, or has aborted due to a problem that is unlikely

Algorithm 3 P8TM: complete algorithm

```
1: Shared variables:
2:     status[N] ← {⊥, ⊥, ..., ⊥}        ▷ One per thread
3:     glock ← FREE                        ▷ Spin lock to serialize transactions
4: Local variables:
5:     tid ∈ [0..N]                        ▷ Identifier of current thread
6:     mode ∈ {HTM, ROT, GL}               ▷ Transaction mode
7:     rot-rset ← ∅                        ▷ Transaction's read-set

8: function BEGIN_W
9:     wait until glock = FREE             ▷ GL must be free
10:    BEGIN_HTM()                         ▷ Try HTM first
11:    if mode ≠ HTM then                  ▷ If HTM fails...
12:        BEGIN_ROT()                     ▷ ...fall back to ROT
13:        if mode ≠ ROT then              ▷ If ROT also fails...
14:            BEGIN_GL()                  ▷ ...default to global lock

15: function BEGIN_HTM
16:    trials ← 0
17:    repeat                              ▷ Retry HTM a few times
18:        trials ← trials+1
19:        tx ← tx_begin                   ▷ HTM begin
20:        if tx = STARTED then            ▷ Success?
21:            if glock ≠ FREE then        ▷ Add lock to read-set
22:                tx_abort                ▷ Abort if lock taken
23:            mode ← HTM                  ▷ Run in HTM mode
24:    until mode = HTM                    ▷ Repeat until success...
       ∨ tx = CAPACITY-ABORT               ▷ ...or capacity abort...
       ∨ trials > MAX-HTM-TRIALS           ▷ ...or too many trial

25: function BEGIN_ROT
26:    trials ← 0
27:    repeat                              ▷ Retry ROT a few times
28:        trials ← trials+1
29:        status[tid] ← ACTIVE            ▷ Update status
30:        MEM_FENCE                       ▷ Make sure others know
31:        if glock ≠ FREE then            ▷ Global lock taken?
32:            status[tid] ← ⊥             ▷ Yes: defer to GL...
33:            wait until glock = FREE     ▷ ...wait...
34:            go to 29                    ▷ ...and retry
35:        rot-rset ← ∅                    ▷ Clear read-set
36:        tx ← tx_begin_rot               ▷ HTM ROT begin
37:        if tx = STARTED then            ▷ Success?
38:            mode ← ROT                  ▷ Run in ROT mode
39:    until mode = ROT                    ▷ Repeat until success...
       ..... ∨ tx = CAPACITY-ABORT         ▷ ...or capacity abort...
       ..... ∨ trials > MAX-ROT-TRIALS     ▷ ...or too many trial

40: function BEGIN_GL
41:    status[tid] ← ⊥                     ▷ Not using TM
42:    MEM_FENCE                           ▷ Make sure others know
43:    repeat                              ▷ Acquire global lock
44:        wait until glock = FREE         ▷ Test and...
45:    until CAS(glock, FREE, LOCKED)      ▷ ...test and set
46:    mode ← GL                           ▷ Run in GL mode
47:    SYNCHRONIZE()                       ▷ Perform quiescence phase

48: function READ(addr)                    ▷ Read shared variable
49:    if mode = ROT then
50:        rot-rset ← rot-rset ∪{addr}     ▷ Track ROT reads

51: function COMMIT_W
52:    switch mode do
53:        case HTM
54:            tx_suspend                  ▷ Suspend transaction
55:            SYNCHRONIZE()               ▷ Perform quiescence phase
56:            tx_resume                   ▷ Resume transaction
57:            tx_commit                   ▷ End transaction
58:        case ROT
59:            tx_suspend                  ▷ Suspend transaction
60:            status[tid] ← ROT-COMMITTING ▷ Tell others...
61:            MEM_FENCE                   ▷ ...we are committing
62:            tx_resume                   ▷ Resume transaction
63:            SYNCHRONIZE()               ▷ Quiescence inside ROT
64:            TOUCH_VALIDATE()            ▷ Touch to validate
65:            tx_commit                   ▷ End transaction
66:            status[tid] ← ⊥
67:        case GL
68:            glock ← FREE                ▷ Release global lock

69: function SYNCHRONIZE
70:    s[N] ← status                       ▷ Read and copy all status variables
71:    for i ← 0 to N−1 do                 ▷ Wait until all threads...
72:        if s[i] = ACTIVE                ▷ ...that are active...
       ..... ∨ (mode = GL ∧ s[i] = ROT-COMMITTING) then
73:            wait until status[i] ≠ s[i] ▷ ...cross barrier

74: function TOUCH_VALIDATE
75:    for addr ∈ rot-rset do              ▷ Re-read all elements...
76:        read ∗addr                      ▷ ...from read-set
```

(e.g., contention) or likely (capacity) to be encountered again in a subsequent attempt.

Transactions try to run in HTM and ROT modes a limited number of times, switching immediately if the cause of the failure is a capacity abort (Lines 24 and 39). The GL fallback uses a basic spin lock, which is acquired upon transaction begin (Lines 43–44) and released upon commit (Line 68). Observe that the quiescence mechanism must also be called after acquiring the lock to wait for completion of ROTs that are in progress and might otherwise see inconsistent updates (Line 47), and that the GL path must actually wait for ROTs to fully complete, not just enter the commit phase as for the the other execution modes (Line 72). The rest of the algorithm is relatively straightforward.

To understand the intuition behind how URO path can be integrated safely with concurrent update transactions, consider first that transactions that do not modify shared data cannot modify the state witnessed by a HTM transaction or a ROT. Furthermore, because HTM transactions and ROTs buffer their writes and quiesce before committing, they cannot propagate inconsistent updates to UROs.

Finally, GL and UROs cannot conflict with each other as long as they do not run concurrently. This is ensured by the quiescence phase after acquiring the global lock, and the fact that UROs do not start executing until the lock is free (Line 6). Note that, if the lock is taken, UROs defer to the update transaction holding the global lock by resetting their status (Line 5) before waiting for the lock to be free and retrying the whole procedure. Otherwise we could run into a deadlock situation with a URO waiting for the lock held by a GL transaction, while the latter is blocked in quiescence waiting for the former to complete its execution.

Algorithm 4 — P8TM: URO path

```
1:  function BEGIN_RO
2:      status[tid] ← ACTIVE        ▷ Update thread's status
3:      MEM_FENCE                   ▷ Ensure visibility to update txs.
4:      if glock ≠ FREE then        ▷ Global lock taken?
5:          status[tid] ← ⊥         ▷ Yes: defer to GL...
6:          wait until glock = FREE ▷ ...wait...
7:          go to 2                 ▷ ...and retry
8:  function COMMIT_RO
9:      MEM_FENCE                   ▷ Avoid re-ordering.
10:     status[tid] ← ⊥             ▷ Reset thread's status
```

5.4 Correctness

In this section we present informal arguments on the safety guarantees provided by P8TM in the presence of concurrent[2] transactions and assuming a data race-free model, i.e., accesses to shared data are only performed from within transactions. Specifically, we show that P8TM guarantees opacity [22].

When the GL path is active, concurrency is disabled. This is guaranteed since: *(i)* transactions in HTM path subscribe eagerly to the GL, and are thus aborted upon the activation of this path; *(ii)* after the GL is acquired, a quiescence phase is performed to wait for active ROTs or UROs (and both are not allowed to start if the lock is held).

Correctness of a transaction in the HTM path is provided by the hardware against concurrent HTM transactions/ROTs and by the eager GL subscription.

As for the UROs, the quiescence mechanism guarantees two properties:

- UROs activated after the start of an update transaction T, and before the start of T's quiescence phase, can be safely serialized before T because they are guaranteed not to see any of T's updates, which are only made atomically visible when the corresponding HTM transaction/ROT commits;
- UROs activated after the start of the quiescence phase of an update transaction T can be safely serialized either before T because they are guaranteed to abort T in case they read a value written by T before T commits, or after T as they will see all the updates produced by T's commit. It is worth noting here though that this is only relevant when a URO may conflict with T, in case of disjoint operation both serialization orders are equivalent.

Now we are only left with transactions running on the ROT path. The same properties of quiescence for UROs apply here and avoid ROTs reading inconsistent states produced by concurrent HTM transactions. Nevertheless, since ROTs do modify the shared state, they can still produce non-serializable histories; such as the scenario in Figure 1. Assume a ROT, say T_1, issued a read on X, developing a read-write conflict, with some concurrently active ROT, say T_2. There are two cases to consider: T_1 commits before T_2, or vice versa.

If T_1 commits first, then if it reads X after T_2 (which is still active) wrote to it, then T_2 is aborted by the hardware conflict detection mechanism. Else, we are in the presence of a write-after-read conflict. T_1 finds

[2] Two transactions are concurrent if the begin call of either of them happens in real time between the begin and commit calls of the other.

$status[T_2] := ACTIVE$ (because T_2 issues a fence before starting) and waits for T_2 to enter its commit phase (or abort). Then T_1 executes its T2V, during which, by re-reading X, would cause T_2 to abort.

Consider now the case in which T_2 commits before T_1. If T_1 reads X, as well as any other memory position updated by T_2, before T_2 writes to it, then T_1 can be safely serialized before T_2 (as T_1 observed none of T_2's updates). If T_1 reads X, or any other memory position updated by T_2, after T_2 writes to it and before T_2 commits, then T_2 is aborted by the hardware conflict detection mechanism; a contradiction. Finally, it is impossible for T_1 to read X after T_2 commits: in fact, during T_2's commit phase, T_2 must wait for T_1 to complete its execution; hence, T_1 must read X after T_2 writes to it and before T_2 commits, falling in the above case and yielding another contradiction.

Finally, because update transactions either execute in hardware (HTM or ROT) or in a global lock, it is guaranteed that there are no side effects for aborted transactions. For transactions executing using HTM and ROT, this is guaranteed by the hardware which buffers all the updates and discards them upon abort (as mentioned in Section 3). For transactions that acquire the global lock, it is guaranteed by the fact that transactions run solo.

5.5 Progress and Fairness

The HTM implementation of POWER8 (as most of commercially available HTM implementations) only provide best-effort progress guarantees, i.e., transactions may never commit in hardware. Similar to other HTM-based systems, P8TM relies on a software fallback to guarantees liveness. The software fallback of P8TM is lock-based, therefore, P8TM inherits the progress and fairness guarantees of the lock being used. In our implementation we used a simple spinlock, which as we will see in Section 8, works well in practice across a wide range of workloads and benchmarks. It is worth noting, though, that a simple spin lock may cause read-only transactions executing in the URO path to starve. This can happen since UROs defer to update transactions when they find the lock busy. Thus, in the presence of a stream of update transactions acquiring the lock, UROs may never get to execute. To solve this issue, one could rely on a versioned lock that is incremented every time it is acquired — a technique used, e.g., for fair implementations of read-write locks [27]. With this approach, when a URO starts, after advertising its status, it reads the lock's version and stores it locally. If the URO finds the lock acquired, it must wait until either the lock is released or the lock's version becomes larger than the one initially observed by the URO. When a new update transaction T acquires the lock, it can only start after all the active UROs that have read an older version of the lock have finished, i.e., all UROs that started before the lock was acquired by T. This avoids the starvation of UROs as they will only defer to at most one update transaction.

6 Self-tuning

In workloads where transactions fit the HTM's capacity limitations, P8TM still forces HTM transactions to incur the overhead of suspend/resume, in order to synchronize them with possible concurrent ROTs. In these workloads, the ideal decision would be to just disable the ROT path, so to spare the HTM path from any overhead. However, it is not trivial to determine when it is beneficial to do so; this decision is workload dependent and it can be hard to determine via static analysis, especially in applications that make intensive use of pointers.

We address this issue by integrating into P8TM a self-tuning mechanism based on a lightweight reinforcement learning technique, UCB [29], which we shall describe shortly. UCB determines, in an automatic fashion, which of the following modes to use:

- M1: HTM falling back to ROT, and then to GL;
- M2: HTM falling back directly to the GL;
- M3: starting directly in ROT before falling back to the GL.

Note that UROs and ROTs impose analogous overheads to HTM transactions. Thus, in order to reduce the search space to be explored by the self-tuning mechanism, whenever the ROT path is disabled, the URO path is also disabled. In such cases read-only transactions are treated as update transactions.

Figure 5 shows the three paths and the rules for switching between them. When ROTs are disabled (M1⇒M2), the quiescence call within transactions can be skipped only once there are no more active ROTs. When switching from M2 to M3, instead, it is enough to enable ROTs after having ensured, via a memory fence, that all threads are informed about the need to enable quiescence. This forces any active HTM transaction to perform the quiescence once it reaches its commit phase, while it will abort any active transaction that has reached commit stage and has not yet committed (since the flag is already part of the read-set). The other rules are straightforward.

UCB. Upper confidence bounds (UCB) [3] is a provably optimal solution to the multiarmed bandit problem [4],

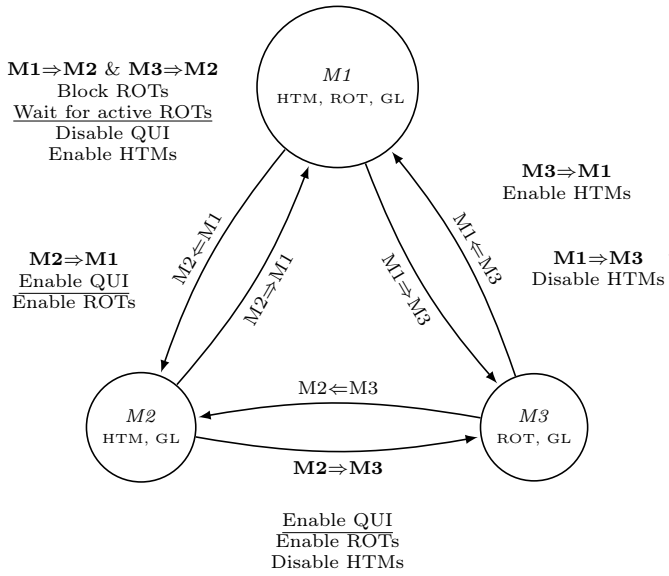

Fig. 5 Different execution paths that can be used by transactions and rules for switching between them. Lines within rules represent necessary memory barriers.

i.e., a classical reinforcement-learning problem that embodies the trade-off between exploration and exploitation. In the multiarmed bandit problem, a gambler tries to maximize the reward obtained from playing different levers of a multiarm slot machine, where the levers' rewards are random variables with a priori unknown distributions. After an initial phase, in which every lever is sampled once, UCB estimates the expected reward for lever i as $\bar{x}_i + \sqrt{(2\log n)/n_i}$, where: $\bar{x}_i$ is the average reward for lever i; n is the number of the current trial; and n_i is the number of times the lever i has been tried.

In order to use UCB in P8TM, we associate each execution mode to a different lever, and its reward to the throughput obtained by using that mode during a sampling interval of 100 microseconds.

We opted for using the UCB technique given that it provides strong theoretical guarantees[3] while imposing negligible computational overheads. Another key advantage of UCB is its generic, parameter-free, black box nature, which makes it not only easy to use/fully automatic, but also robust to architectural changes (e.g., deployments on alternative architectures equipped with a different number of slower/faster cores). This is in contrast with domain specific, threshold-based heuristics [31, 37, 39] that determine the strategy to adopt based on on-line gathered statistics, such as abort rate, frequency of capacity exception, ratio of read-only versus update transactions. In fact, the effectiveness of the latter approaches hinges on the correct tuning of the threshold values that are used to decide which modes to adopt - a non-trivial problem which is sometimes approached by developing ad-hoc analytical models, e.g., [11, 41], and that can be totally circumvented thanks by adopting a pure black-box approach such as UCB.

A noteworthy limitation of the UCB method is that it assumes that the distribution of the levers (i.e., the workload) to be static, i.e., not to vary over time. In practice, this implies that if the workload changes after a long time, UCB is likely to react slowly to changes. In fact, in such settings, UCB will tend to stick with the "old" optimal configuration and seldom test alternative configurations. As a consequence, UCB may fail to detect the emergence of a different optimum strategy in a prompt fashion. Note that more sophisticated variants of UCB do exist that detect statistically signifying changes in the levers' distribution [20, 38, 47], but they typically introduce additional parameters (e.g., to control the reactivity of the self-tuning scheme) that require appropriate tuning — thus loosing one of the key appealing features of UCB, namely its parameter-free nature. As we shall see in Section 8, though, the UCB method works pretty well in practice, despite its simplicity. In fact, even in benchmarks whose workloads is known to shift over time, such as Genome [35], the UCB-based approach improves consistently over a static approach that always operates according to mode M1 (see above).

7 Read-set Tracking

The T2V mechanism requires to track the read-sets of ROTs for later replaying them at commit time. The implementation of the read-set tracking scheme is crucial for the performance of P8TM. In fact, as discussed in Section 3, ROTs do not track loads at the TMCAM level, but they do track stores and the read-set tracking mechanism must issue stores in order to log the addresses read by a ROT. The challenge, hence, lies in designing a software mechanism that can exploit the TMCAM's capacity in a more efficient way than the hardware would do. In the following we describe two alternative mechanisms that tackle this challenge by exploring different trade-offs between computational and space efficiency.

Time-efficient implementation uses a thread-local cache-aligned array, where each entry is used to track a 64-bit address. Since the cache lines of the POWER8 CPU are 128 bytes long, this means that 16 consecutive entries of the array, each storing an arbitrary address, will be mapped to the same cache line and occupy a single TMCAM entry. Therefore, this approach allows

[3] Logarithmic bounds on the cumulative error, called regret, from playing non-optimal levers even in finite time horizons [3].

for fitting up to 16× larger read-sets within the TMCAM as compared to the case of HTM transactions. Given that they track 64 cache lines, thread-local arrays are statically sized to store exactly 1024 addresses. It is worth noting here that since conflicts are detected at the cache line level granularity, it is not necessary to store the 7 least significant bits, as addresses point to the same cache line. However, we omit this optimization as this will add extra computational overhead, yielding a space saving of less than 10%.

Space-efficient implementation seeks to exploit the spatial data locality in the application's memory access patterns to compress the amount of information stored by the read-set tracking mechanism. This is achieved by detecting a common prefix between the previously tracked address and the current one, and by storing only the differing suffix and the size (in bytes) of the common prefix. The latter can be conveniently stored using the 7 least significant bits of the suffix, which, as discussed, are unnecessary. With applications that exhibit high spatial locality (e.g., that sequentially scan memory), this approach can achieve significant compression factors with respect to the time-efficient implementation. However, it introduces additional computational costs, both during the logging phase (to identify the common prefix) and in the replay phase (as addresses need to be reconstructed).

8 Evaluation

In this section we evaluate P8TM against state-of-the-art TM systems using a set of synthetic micro-benchmarks and complex, real-life applications. First, we start by evaluating both variants of read-set tracking to show how they are affected by the size of transactions and degree of contention. Then we conduct a sensitivity analysis aimed to investigate various factors that affect the performance of P8TM. To this end, we used a micro-benchmark that manipulates a hashmap via lookup, insert, and delete transactions. Finally, we test P8TM using two complex, realistic workloads: the popular STAMP benchmark suite [35] and a port to the TM domain of the TPC-C benchmark for in-memory databases [18].

We compare our solution with the following baselines: *(i)* plain HTM with a global lock fallback (HTM-SGL), *(ii)* NOrec with write back configuration, *(iii)* the Hybrid NOrec algorithm with three variables to synchronize transactions and NOrec fallback, and finally, *(iv)* the hardware read-write lock elision algorithm HERWL [19], where we have update transactions acquiring the write lock and read-only transactions acquiring the read lock. To demonstrate how P8TM fares against non-TM solutions, we added RCU as another baseline in the sensitivity study. Note that it is hard to devise a hand-tuned concurrency implementation for STAMP and TPC-C benchmarks.

Regarding the retry policy, we execute HTM path 10 times and the ROT path 5 times before falling back to the next path, except upon a capacity abort when the next path is directly activated. These values and strategies were chosen after doing an extensive offline experiment and selecting the best on average with different number of retries and different capacity aborts handling policies (e.g., fallback immediately vs. treating it as a normal abort). All results presented in this section represent the mean value of at least 5 runs. The experiments were conducted on a machine equipped with IBM POWER8 8284-22A processor that has 10 physical cores, with 8 hardware threads each.

The source code, which is publicly available [25], was compiled with GCC 6.2.1 using `-O2` flag on top of Fedora 24 with Linux 4.5.5. Thread pinning was used to pin a thread per core at the beginning of each run for all the solutions, and threads were distributed evenly across the cores.

8.1 Read-set Tracking

The goal of this section is to understand the trade-off between the time-efficient and the space-efficient implementations of read-set tracking that were explained earlier in Section 7. we compare three variants of P8TM: *(i)* a time-efficient read-set tracking (TE), *(ii)* a variant of space-efficient read-set tracking that only checks for prefixes of length 4 bytes, and otherwise stores the whole address (SE), and, finally, *(iii)* a more aggressive version of space-efficient read-set tracking that looks for prefixes of either 6 or 4 bytes (SE++).

Throughout this section, we fixed the number of threads to 10 (number of physical cores) and the percentage of update transactions at 100%, disabled the self-tuning module, and varied the transaction length across orders of magnitude to stress the ROT-path.

First, we consider an almost contention-free workload to highlight the effect of capacity aborts alone. The speed-up with respect to HTM-SGL, breakdown of causes of aborts and commits for this workload is shown in Figure 6. For the breakdown of causes of aborts, we report the following: *(i)* conflict aborts between HTM transactions (HTM tx); *(ii)* conflict aborts between HTM transactions and non-transactional code (HTM non-tx); *(iii)* aborts due to exceeding the capacity limit of HTM transactions (HTM capacity); *(iv)* aborts triggered when the lock is found held after the beginning

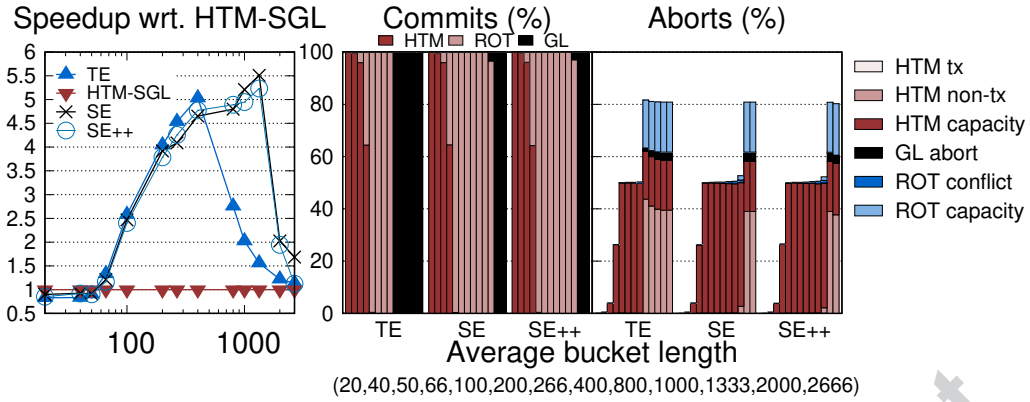

Fig. 6 Evaluation of different implementations of read-set tracking in the absence of contention.

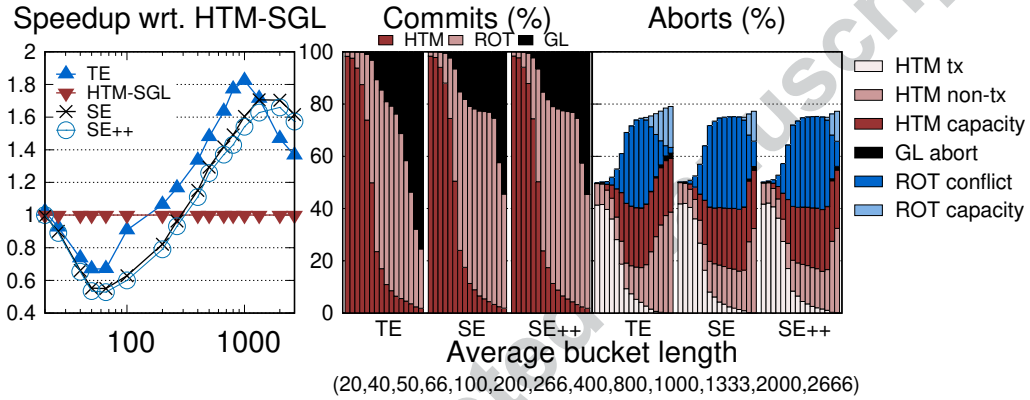

Fig. 7 Evaluation of different implementations of read-set tracking in the presence of contention.

of HTM transactions (GL Abort); *(v)* conflict aborts between ROT and (non-)transactional code (ROT conflict) *(vi)* aborts due to exceeding the capacity limit of ROTs (ROT capacity). For the breakdown of commit modes, we report the following: *(i)* commits using HTM transactions (HTM); *(ii)* commits using ROTs (ROT); *(iii)* commits using global lock (GL).

As we can notice, the three variants of P8TM achieve almost the same performance as HTM-SGL with small transaction sizes that fit inside regular HTM transactions, as seen from the speed-up. However, when moving to larger transactions, the three variants start outperforming HTM-SGL by up to 5.5× due to their ability to fit transactions within ROTs. By looking at the aborts breakdown with larger transactions, we see that all P8TM variants suffer from almost 50% capacity aborts when first executing in HTM, and almost no capacity aborts when using the ROT path. This shows the clear advantage of the T2V mechanism and how it can fit more than 10× larger transactions in hardware.

Comparing TE with SE and SE++, we see that both space-efficient variants are able to execute even larger transactions as ROTs. Nevertheless, they incur an extra overhead, which is reflected as a slightly lower speed-up than TE, before this starts to experience ROT capacity aborts; only then their ability to further compress the read-set within TMCAM pays off. Again, by looking at the commits and aborts breakdown, we see that both space-efficient variants manage to commit all transactions as ROTs when TE is already forced to execute using the GL. Finally, when comparing SE and SE++, we notice that trying harder to find longer prefixes is not useful, as in this workload there is a very low probability that the accessed addresses share 6 bytes long prefixes.

Figure 7 shows the results for a workload that exhibits a higher degree of contention. In this case, with transactions that fit inside regular HTM transactions, we see that HTM-SGL can outperform both SE and SE++ by up to 2× and TE by up to ∼30%. Since P8TM tries to execute transactions as ROTs after failing 10 times with HTM due to conflicts, the ROT path may be activated even in the absence of capacity aborts; hence, the overhead of synchronizing ROTs and HTM transaction becomes relevant even with small transactions. With larger transactions, we notice that the computational costs of SE and SE++ are more noticeable in this workload where they are always outperformed by TE, as long as this is able to fit at least 50% of transactions inside

ROTs. Furthermore, the gains of SE and SE++ with respect to TE are much lower when compared to the contention-free workload. From this, we deduce that TE is more robust to contention. This was also confirmed with the other workloads that will be discussed next.

8.2 Sensitivity Analysis

We now report the results of a sensitivity analysis that aimed to assess the impact of the following factors on P8TM's performance: *(i)* the size of transactions, *(ii)* the degree of contention, and *(iii)* the percentage of read-only transactions. We explored these three dimensions using the following configurations: *(i)* high capacity, low contention, *(ii)* high capacity, high contention, and *(iii)* low capacity, low contention, with 10%, 50%, and 90% update transactions. We do not report the results for low capacity, high contention workload, since they do not convey any extra information with respect to the low capacity, low contention scenario (which is actually even more favorable for HTM).

To design these workloads we set the number of buckets of the hashmap to 10 for high contention scenario and 1000 for low contention scenario. The total number of items in the hashmap, which determines the length of each bucket controls the degree at which capacity aborts are being triggered. For the scenario with low probability of triggering capacity aborts (∼2%), we set the total number of items such that each bucket has 50 items. Whereas, for the workload with high probability (∼50%) of triggering capacity aborts, we set the total number of items such that each bucket has 500 items. To maintain a fixed size of the hashmap throughout the experiment, update transactions alternate between insert and delete operations. All three types of transactions, insert, delete and lookup access the hashmap following a uniform access pattern.

In these experiments we show two variants of P8TM, both equipped with the TE read-set tracking: with (P8TM^{UCB}) and without (P8TM) the self-tuning module enabled.

High capacity, low contention. Figure 8 shows the throughput, as well as the breakdown of abort causes and commit modes for the high capacity, low contention configuration. The considered abort causes are similar to the one included in Figure 6 except that for *GL Abort* that is now *GL/STM Abort* to refer the STM aborts for HyNOrec. For the breakdown of commits, we now also show *(i)* commits using the uninstrumented read-only execution path (URO), and *(ii)* commits using either global lock (for HTM-SGL, P8TM and HERWL) or using STM (for HyNOrec) (GL/STM).

We observe that for the read-dominated workload, all variants of P8TM are able to outperform all the other TM solutions by up to 7×. This can be easily explained by looking at the commits breakdown, where P8TM and P8TM^{UCB} commit 90% of their transactions as UROs while the other 10% are committed as ROTs. Compared to RCU, P8TM is capable of achieving ∼3.5× higher throughput. Although RCU is designed for such workloads, the fact that P8TM can executed update transactions concurrently, unlike RCU, allows it to achieve better performance.

On the contrary, HTM-SGL commits only 10% of the transactions in hardware and falls back to GL in the rest, due to the high capacity aborts it incurs. It is worth noting that the decrease in the percentage of capacity aborts, along with the increase of number of threads, is due to the activation of the fallback path, which forces other concurrent transactions to abort.

Although HERWL is designed for such workloads, P8TM was able to achieve ∼2.4× higher throughput, thanks to its ability of executing ROTs concurrently. Another interesting point is that P8TM^{UCB} can outperform P8TM thanks to its ability to decrease the abort rate, as shown in the aborts breakdown. This is achieved by deactivating the HTM path, which spares P8TM from the cost of trying once in HTM before falling back to ROT (upon a capacity abort).

We can see a similar trend when moving to the workloads with more update transactions: P8TM and P8TM^{UCB} outperform HTM-SGL by ∼2.2× and ∼1.4× in the 50% and 90% workloads, respectively. They also achieve the highest throughput in all workloads among all the considered baselines. By looking at the breakdown of commits, we can see that P8TM executes almost all update transactions using either HTMs or ROTs up to 8 threads, unlike HTM-SGL that only executes 10% of transactions in hardware. At high thread count we notice that NOrec and HyNOrec start to outperform both P8TM and P8TM^{UCB}, especially in the 90% workload. This can be explained by two reasons: *(i)* with larger numbers of threads there is higher contention on hardware resources (note that starting from 32 threads ROT capacity aborts start to become frequent) and *(ii)* the cost of quiescence becomes more significant as threads have to wait longer. Despite that, P8TM variants achieve 2× and ∼1.4× higher throughput than NOrec and HyNOrec, when comparing their maximum throughput regardless of the thread count. RCU is designed for read-dominated workloads, hence, its poor performance in update-intensive workloads as it allows a single writer at a time, unlike TM-based solutions

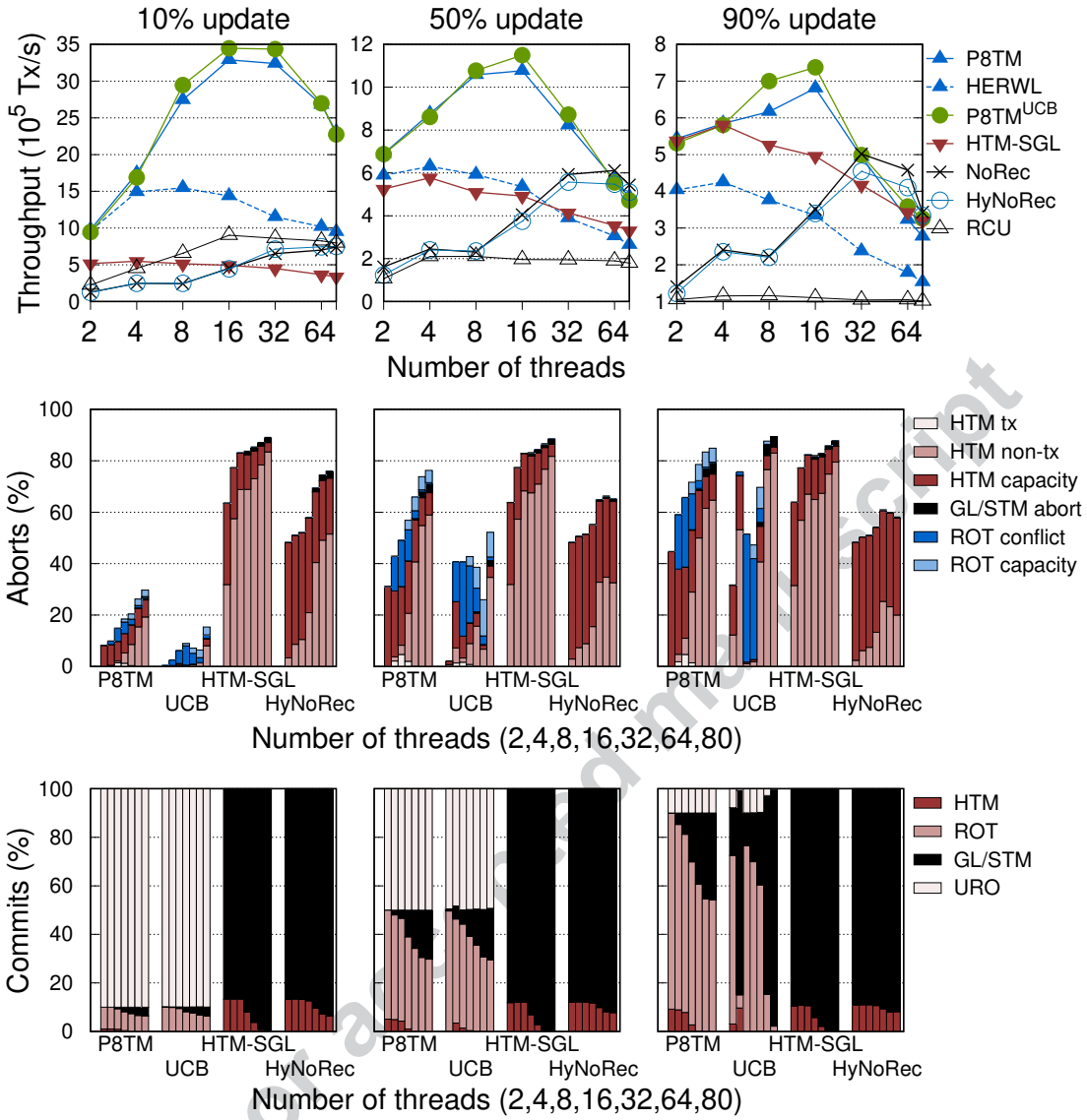

Fig. 8 High capacity-low contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

High capacity, high contention. Figure 9 reports the results for the high capacity, high contention configuration. Trends for read-dominated workload are similar to the case of lower contention degree. However, scalability is much lower here due to the higher conflict rate. When considering the workloads with 50% and 90% update transactions, we notice that P8TM still achieves the highest throughput. Moreover, unlike in the low contention scenario, P8TM outperforms both NOrec and HyNOrec even at high thread count. This is due to the fact that handling contention is more efficiently done at the hardware level than in software.

Although HERWL uses URO to execute read-only transactions, it was unable to scale even in the 90% read-only workload, where its throughput is more than 2× lower than P8TM's. Again this is due to its inability to execute concurrent ROTs. This clearly indicates that T2V is beneficial even in read-dominated workloads. In such workloads, however, we can see that RCU achieves better performance than HERWL as RCU's pessimistic execution path for update transactions avoids wasting time with speculative execution. Nevertheless, it still yields lower throughput when compared to P8TM.

Low capacity, low-contention. In workloads where transactions fit in HTM, it is expected that HTM-SGL will outperform all other TM solutions and that the overheads of P8TM will prevail. Results in Figure 10 confirm this expectation: HTM-SGL outperforms all other solutions, regardless of the ratio of read-only transactions,

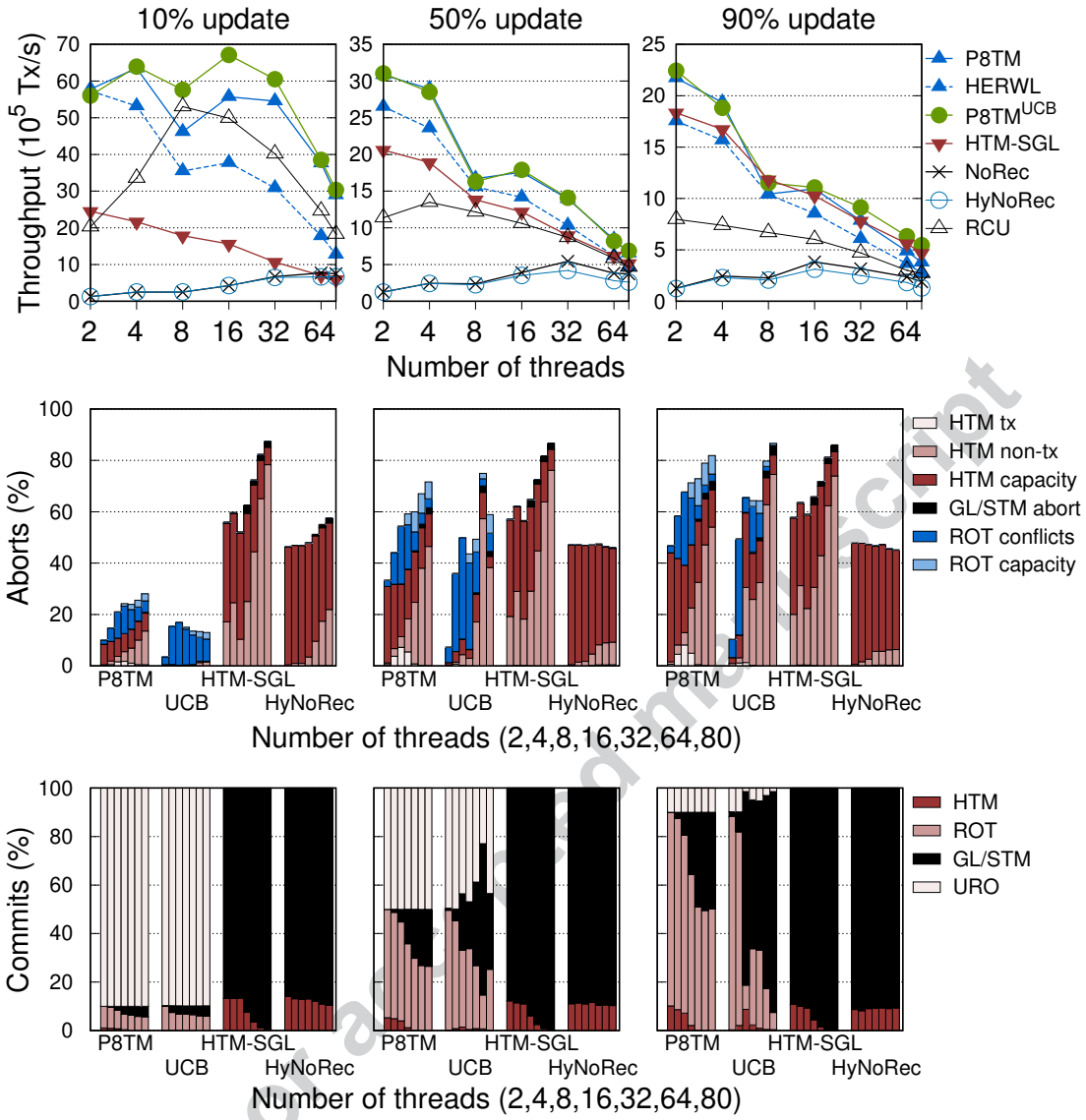

Fig. 9 High capacity, high contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

achieving up to 2.5× higher throughput than P8TM. However, P8TM^{UCB}, thanks to its self-tuning ability, is the, overall, best performing solution, achieving performance comparable to HTM-SGL at low thread count, and outperforming all other approaches at high thread count. By inspecting the commits breakdown plots we see that P8TM^{UCB} does not commit any transaction using ROTs up to 8 threads, avoiding the synchronization overheads that, instead, affect P8TM.

It is worth noting, though, that P8TM, with 90% read-only transactions, does outperform HTM-SGL beyond 16 threads. By inspecting the breakdown of aborts and commits we notice that when hardware multithreading is enabled the performance of HTM-SGL deteriorates dramatically, due to the increased contention on hardware resources. Conversely, P8TM can still execute transactions in UROs/ROTs, hence achieving higher throughput.

We note that, even though HyNOrec commits the same or higher percentage of HTM transactions than HTM-SGL, it is consistently outperformed by P8TM. This can be explained by looking at the performance of NOrec, which fails to scale due to the high instrumentation overheads it incurs with such short transactions. As for HyNOrec, its poor performance is a consequence of the inefficiency inherited by its NOrec fallback.

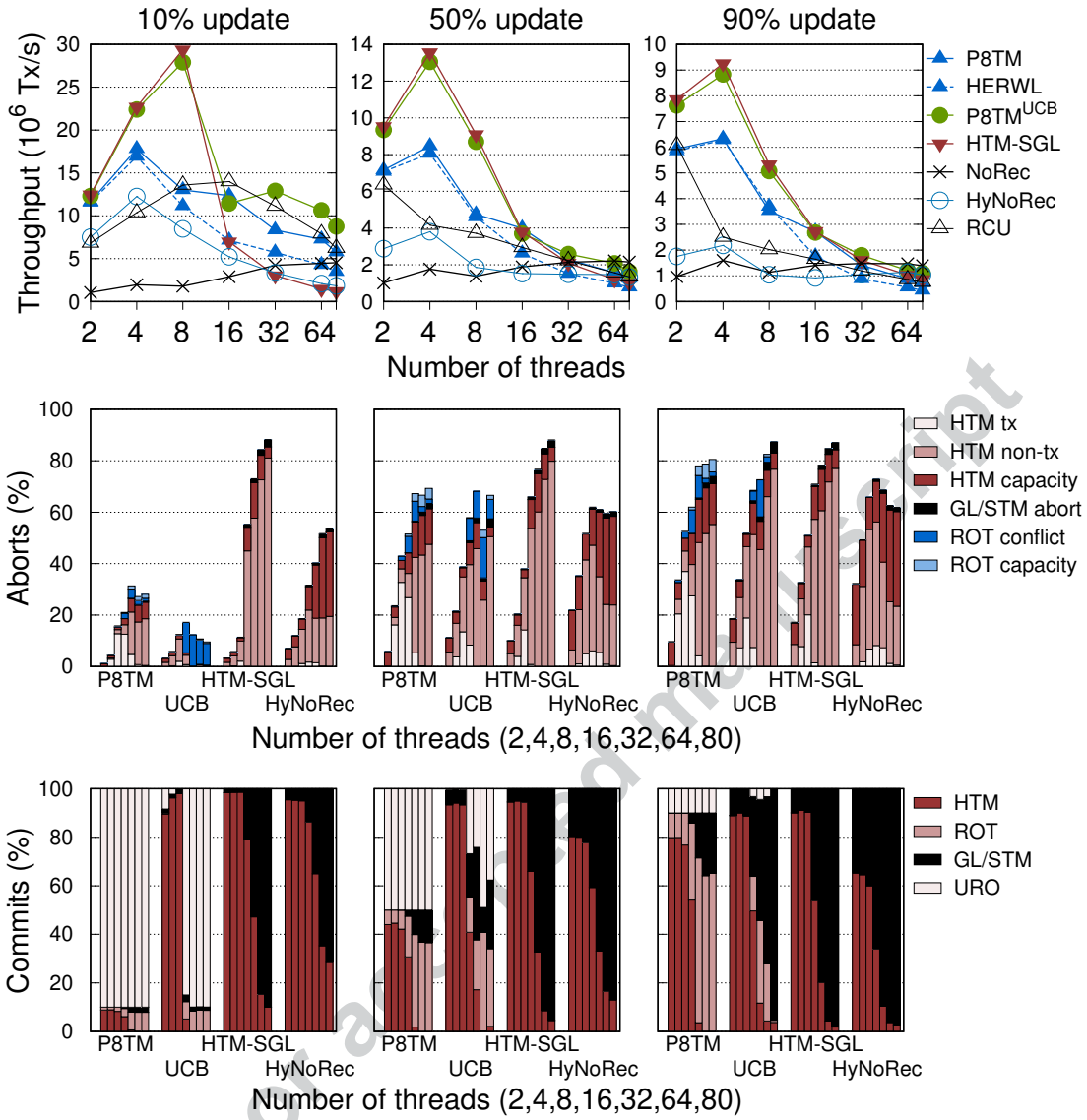

Fig. 10 Low capacity, low contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

8.3 STAMP Benchmark Suite

STAMP is a popular benchmark suite in the TM domain, that encompasses applications with different characteristics that share a common trait: they do not have any read-only transactions. Therefore, P8TM will not utilize the URO path and any gain it can achieve stems solely from executing ROTs in parallel. We omitted the results of the Bayes benchmark due to its high variance [45]. Labyrinth is also omitted, as its transactions do not fit in neither HTM nor ROT—hence, exhibiting very similar performance trend to Yada.

Genome and **Vacation** are two applications with medium sized transactions and low contention; hence, they behave similarly to the previously analyzed high capacity, low contention workloads. When looking at Figure 11, we can see trends very similar to the workloads with high update ratios in Figure 8. P8TM is capable of achieving the highest throughput and outperforming HTM-SGL by up to 4.5× in case of Genome and ∼3.2× in the case of Vacation. Again P8TM^{UCB} is even able to achieve higher throughput than P8TM due to deactivating the HTM path when capacity aborts are encountered, thus decreasing the abort rate. When looking at the breakdown of commits, we notice also the ability of P8TM to execute most of transactions in either HTM or ROT at low thread counts. One difference between Genome and Vacation is that, in Vacation, HTM-SGL never manages to commit transactions in hardware.

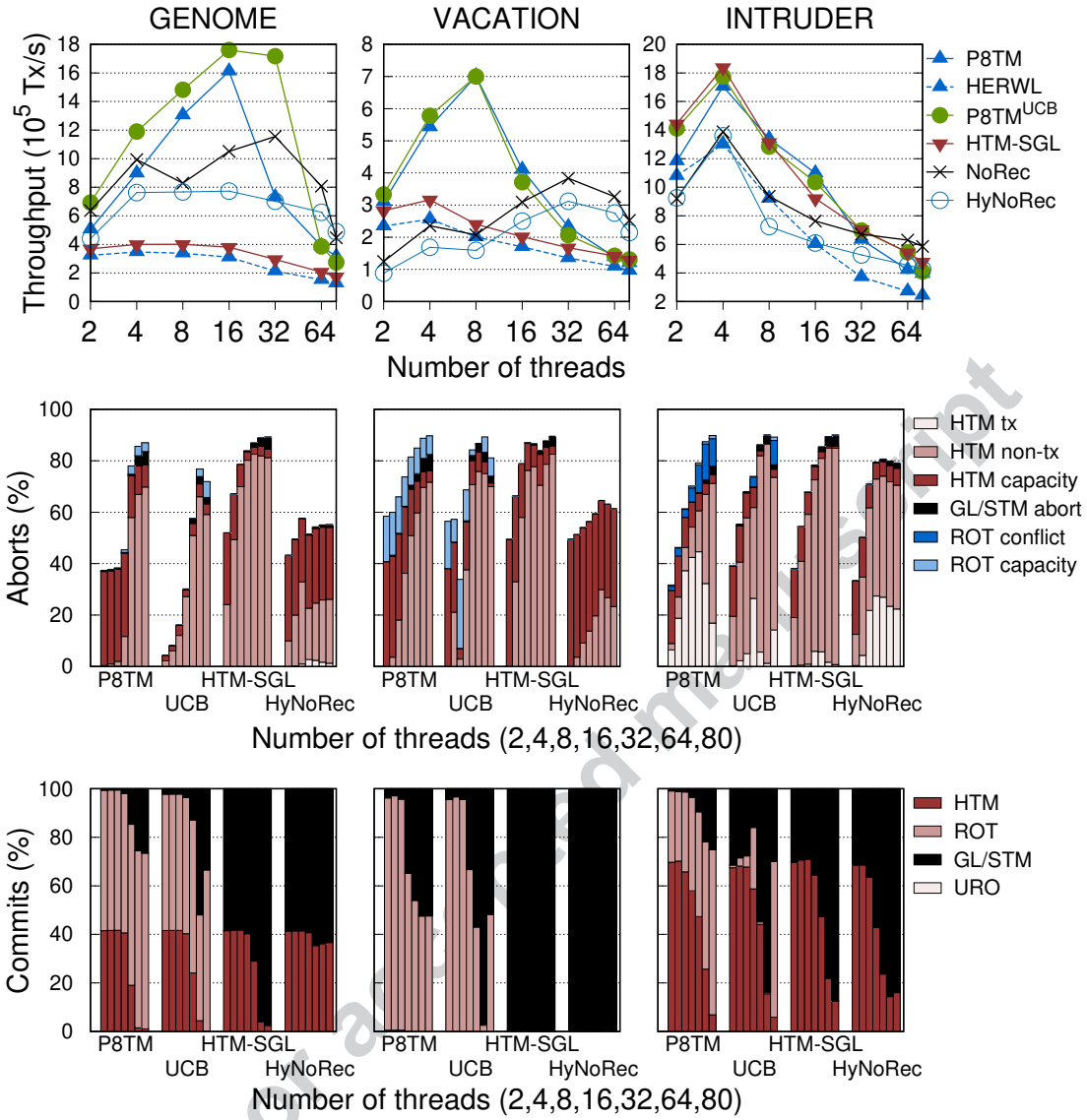

Fig. 11 Throughput, abort rate, and breakdown of commit modes of STAMP benchmarks (1).

We also notice the same drawback at high number of threads when comparing P8TM to NOrec and HyNOrec. Nevertheless, it is worth noting that the maximum throughput achieved by P8TM (at 16 threads) is 1.5× and 2× higher than NOrec (at 32 threads) in Genome and Vacation, respectively. This is due to instrumentation overheads of these solutions. These overheads are completely eliminated in case of write accesses within P8TM and are much lower for read accesses.

Intruder generates transactions with medium read/write sets and high contention. This results in a similar performance for both P8TM and HTM-SGL: they achieve almost the same peak throughput at 8 threads and follow the same pattern with increasing number of threads. Although P8TM manages to execute all transactions as either HTM transactions or ROTs at low numbers of threads, given the low level of parallelism, the synchronization overheads incurred by P8TM are not outweighed by its ability to run ROTs concurrently. Nevertheless, P8TM^{UCB} manages to overcome this limitation by disabling the ROT path and avoid these overheads. Both NOrec and HyNOrec were outperformed, which is again simply due to their high instrumentation costs.

SSCA2 and **KMeans** generate transactions with small read/write sets and low contention. These are HTM friendly characteristics, and by looking at the throughput results in Figure 12 we see that HTM-SGL is able to outperform all the other baselines and scale up to 80 threads in case of SSCA2 and up to 16 threads in case of Kmeans. Although HyNOrec was able to achieve performance similar to HTM up to 32 threads in SSCA2

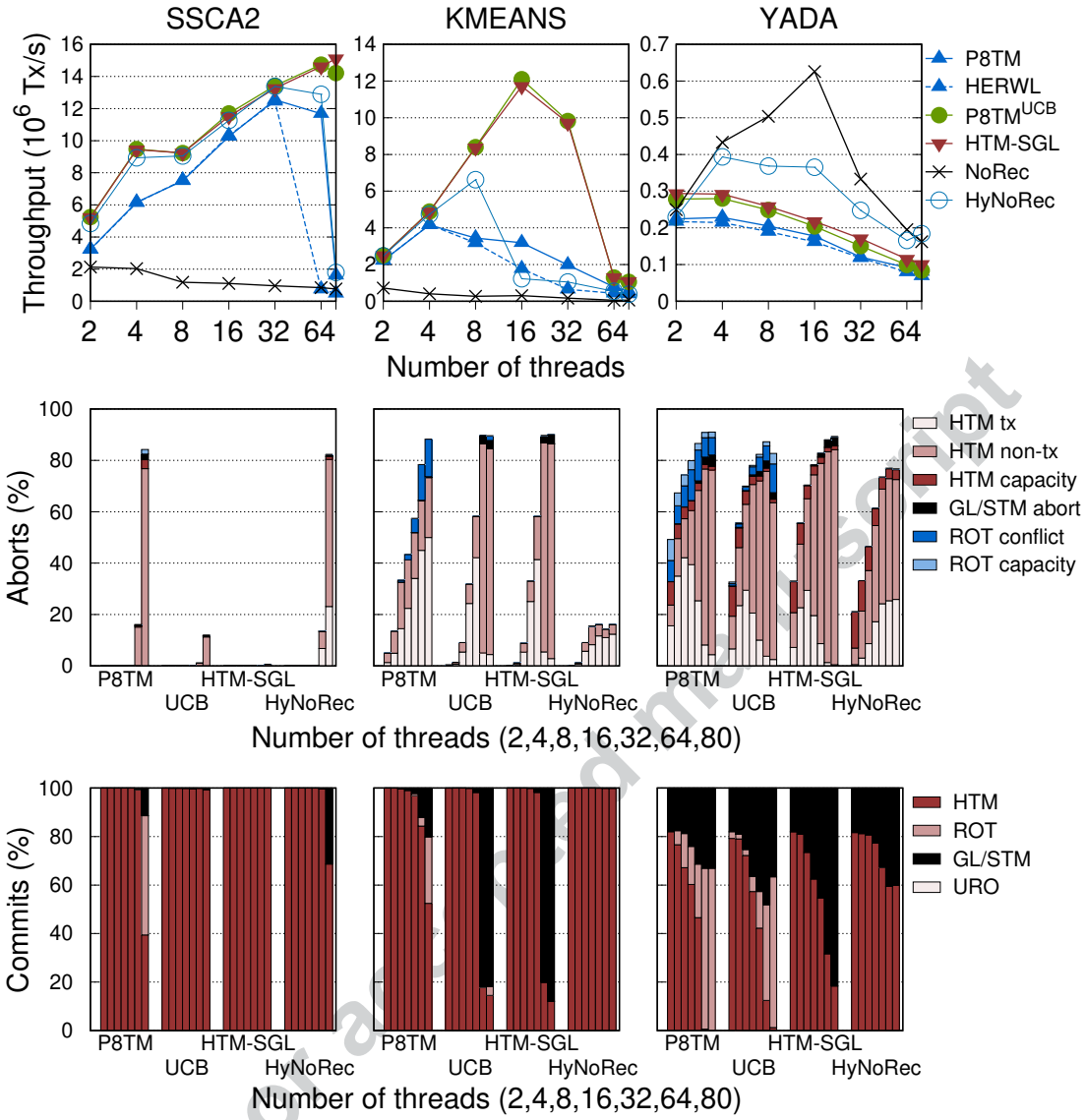

Fig. 12 Throughput, abort rate, and breakdown of commit modes of STAMP benchmarks (2).

and 8 threads in KMeans, it was then outperformed due to the extra overheads it incurs to synchronize with the NOrec fallback. These overheads lead to increased capacity aborts as seen in the aborts breakdown.

Although P8TM commits almost all transactions using HTM up to 64 threads, it performed worse than both HTM-SGL and HyNOrec in SSCA2 due to the costs of synchronization. An interesting observation is that the overhead is almost constant up to 32 threads, since up to 32 threads there are no ROTs running and the overhead of the quiescence call is dominated by the cost of suspending and resuming the transaction. At 64 and 80 threads P8TM started to suffer also from capacity aborts similarly to HyNOrec. This led to a degradation of performance, with HTM-SGL achieving 7× higher throughput at 80 threads. Similar trends can be seen for KMeans, however with different threads counts and with lower adverse effects for P8TM. Again, these are workloads where P8TM^{UCB} comes in handy as it manages to disable the ROT path and thus tends to employ HTM-SGL, which is the most suitable solution for these workloads.

Yada has long transactions, large read/write set and medium contention. This is an example of a workload that is not hardware friendly and where hardware solutions are expected to be outperformed by software based ones. Figure 12 shows the clear advantage of NOrec over any other solution, achieving up to 3× higher throughput than hardware based solutions. When looking at the commits and abort breakdown, one can see that up to 8 threads P8TM commits ∼80% of the transactions as either HTM or ROTs. Moreover, unlike Intruder where

HTM-SGL was able to commit a smaller percentage of transactions in hardware, HTM-SGL is unable to scale with Yada. This can be related to the difference in the nature of workloads, where the transactions that trigger capacity abort form the critical path of execution; hence with such workloads it is not preferable to use hardware-based solutions.

8.4 TPC-C Benchmark

TPC-C represents a wholesale supplier benchmark for relational databases [43]. In this work we use a version ported to work on an in-memory database [18,42], which we adapted to support TM. TPC-C has 5 different types of transactions, two of which are long read-only transactions that have a high chance of generating capacity exceptions. Figure 13 shows the results for workloads with 10%, 50%, and 90% update transactions that consists of a mix of the five types of transactions. For TPC-C, we also report the average latency of both update and read-only transactions. We use the CPU time-stamp counters to measure the time it takes for a transaction to execute, starting from the first begin call until it successfully commits, eventually. Note that we omitted showing latency results for NOrec and HyNOrec to enhance visualization, as they incur significantly higher latency values.[4]

Throughput results show clear advantage of P8TM over all the other baselines in all workloads, regardless of the number of active threads. When compared with software based solutions, P8TM is able to achieve up to 5× higher throughput than both NOrec and HyNOrec at 16 threads in the 90% update workload. Although both NOrec and HyNOrec can scale up to 16 threads, their lower performance can be explained by the much lower instrumentation overheads that P8TM incurs when compared to software-based solutions. When compared to HTM-SGL, P8TM achieves 5.5× higher throughput with workloads that have a high percentage of read-only transactions, thanks to the URO path. When moving to workloads with higher percentages of update transactions, P8TM still outperforms HTM-SGL by 2× and 1.25× on the 50% and 90% update workloads, respectively. Again, looking at the breakdown plots, we can notice that P8TM is able to commit all update transactions either as HTM or ROTs up to 8 threads. We notice that P8TM^{UCB} manages to achieve even further improvement in throughput by disabling the HTM path, hence decreasing the abort rate significantly.

When comparing the latency of read-only transactions, we can notice that in read-dominated workloads, P8TM achieves up to one order of magnitude lower latency than HTM-SGL at 32 threads and an average (geometric mean) latency reduction of ∼6.2× across all thread counts. We can also notice that both P8TM and HERWL consistently achieve, on average, a slightly lower latency for read-only transactions compared to P8TM^{UCB}. This can be attributed to the overhead imposed by P8TM^{UCB} for determining which execution path transaction should follow. For the update transactions, in the read-dominated workload, HTM-SGL achieves up to ∼1.8× lower latency compared to P8TM at 32 threads and an average (geometric mean) latency reduction of 7% across all thread counts. Update transactions with P8TM execute the SYNCHRONIZE() function before committing, which can impose a non-negligible delay if there is a large number of active long-running transactions using the URO path. It is worth recalling here that update transactions do not wait for newly activated read-only transactions, which places an upper bound on the delay and justifies why the latency of update transaction does not grow significantly. Overall, we note that the gains in terms of reduced latency for read-only transactions largely outweighs the slow-down imposed to update transactions, which explains why P8TM and P8TM^{UCB} achieve much higher throughput with respect to HTM-SGL.

Finally, thanks to its ability to choose the most effective execution path, P8TM^{UCB} yields an increased latency for update transactions, when compared to HTM-SGL, only with low number of threads. However, P8TM^{UCB} manages to achieve up to ∼3× lower latency with higher number of threads with respect to both HTM-SGL and P8TM. These gains stem from the fact that P8TM^{UCB} opts for using ROTs for update transactions, thus avoiding wasting time in unsuccessful attempts using HTM.

9 Conclusion

We presented P8TM, a TM system that tackles what is, arguably, the key limitation of existing HTM systems: the inability to execute transactions whose working sets exceed the capacity of CPU caches. This is achieved by novel techniques that exploit hardware capabilities available in POWER8 processors. Via an extensive experimental evaluation, we have shown that P8TM provides robust performance across a wide range of benchmarks, ranging from simple data structures to complex applications, and achieves remarkable speedups.

[4] To minimize instrumentation overhead, latency measurements were only performed on a single thread. Since in this benchmarks all threads execute all type of transactions, this approach does not introduce any bias in the results we gathered.

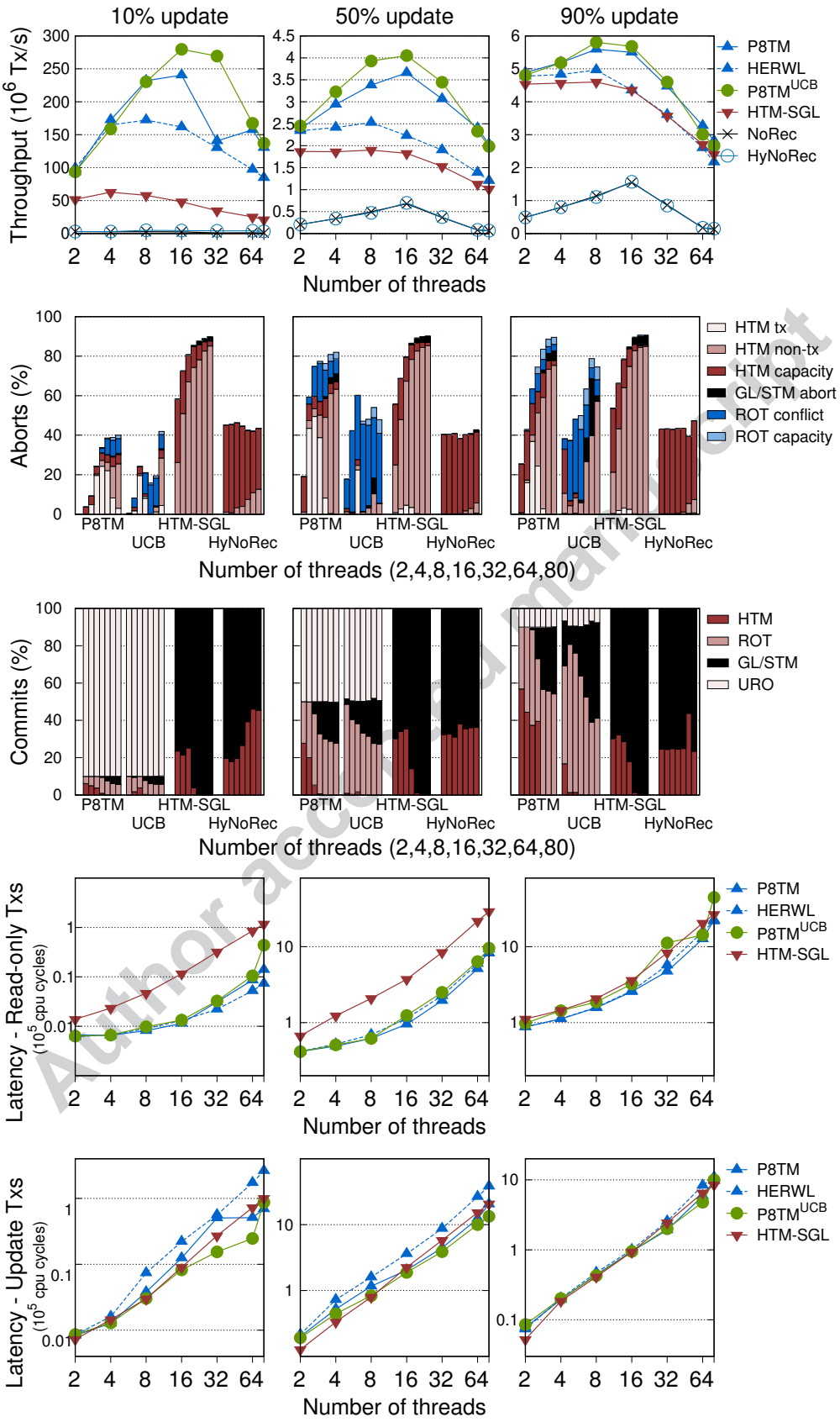

Fig. 13 Throughput, abort rate, breakdown of commit modes and latency of read-only and update transactions of TPC-C at 10%, 50% and 90% update ratios.

The importance of P8TM stems from the consideration that the best-effort nature of current HTM implementations is not expected to change in the near future. Therefore, techniques that mitigate the intrinsic limitations of HTM can broaden its applicability to a wider range of real-life workloads. We conclude by arguing that the performance benefits achievable by P8TM thanks to the use of the ROT and suspend/resume mechanisms represent a relevant motivation for integrating these features in future generations of HTM-enabled processors (like Intel's ones).

Acknowledgements This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2019 and PTDC/EEISCR/1743/2014.

References

1. Afek, Y., Levy, A., Morrison, A.: Programming with hardware lock elision. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, pp. 295–296. ACM, New York, NY, USA (2013). DOI 10.1145/2442516.2442552. URL http://doi.acm.org/10.1145/2442516.2442552
2. Arbel, M., Attiya, H.: Concurrent updates with RCU: Search tree as an example. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC '14, pp. 196–205. ACM, New York, NY, USA (2014). DOI 10.1145/2611462.2611471. URL http://doi.acm.org/10.1145/2611462.2611471
3. Auer, P.: Using confidence bounds for exploitation-exploration trade-offs. J. Mach. Learn. Res. **3**, 397–422 (2003). URL http://dl.acm.org/citation.cfm?id=944919.944941
4. Berry, D., Fristedt, B.: Bandit problems. London: Chapman and Hall (1985)
5. Boehm, H., Gottschlich, J., Luchangco, V., Michael, M., Moir, M., Nelson, C., Riegel, T., Shpeisman, T., Wong, M.: Transactional Language Constructs for C++. ISO/IEC JTC1/SC22 WG21 (C++) (2012)
6. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.: Robust architectural support for transactional memory in the power architecture. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13, pp. 225–236. ACM, New York, NY, USA (2013). DOI 10.1145/2485922.2485942. URL http://doi.acm.org/10.1145/2485922.2485942
7. Calciu, I., Shpeisman, T., Pokam, G., Herlihy, M.: Improved single global lock fallback for best-effort hardware transactional memory. 9th ACM SIGPLAN Wkshp. on Transactional Computing (2014)
8. Clements, A.T., Kaashoek, M.F., Zeldovich, N.: Scalable address spaces using RCU balanced trees. In: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pp. 199–210. ACM, New York, NY, USA (2012). DOI 10.1145/2150976.2150998. URL http://doi.acm.org/10.1145/2150976.2150998
9. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M.L., Spear, M.F.: Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, pp. 39–52. ACM, New York, NY, USA (2011). DOI 10.1145/1950365.1950373. URL http://doi.acm.org/10.1145/1950365.1950373
10. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, pp. 336–346. ACM, New York, NY, USA (2006). DOI 10.1145/1168857.1168900. URL http://doi.acm.org/10.1145/1168857.1168900
11. Di Sanzo, P., Ciciani, B., Palmieri, R., Quaglia, F., Romano, P.: On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. Perform. Eval. **69**(5), 187–205 (2012). DOI 10.1016/j.peva.2011.05.002. URL http://dx.doi.org/10.1016/j.peva.2011.05.002
12. Dice, D., Harris, T.L., Kogan, A., Lev, Y., Moir, M.: Hardware extensions to make lazy subscription safe. CoRR **abs/1407.6968** (2014)
13. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early experience with a commercial hardware transactional memory implementation. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, pp. 157–168. ACM, New York, NY, USA (2009). DOI 10.1145/1508244.1508263. URL http://doi.acm.org/10.1145/1508244.1508263
14. Dice, D., Shavit, N.: Understanding tradeoffs in software transactional memory. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO '07, pp. 21–33. IEEE Computer Society, Washington, DC, USA (2007). DOI 10.1109/CGO.2007.38. URL http://dx.doi.org/10.1109/CGO.2007.38
15. Diegues, N., Romano, P.: Self-tuning intel transactional synchronization extensions. In: 11th International Conference on Autonomic Computing (ICAC 14), pp. 209–219. USENIX Association, Philadelphia, PA (2014). URL https://www.usenix.org/conference/icac14/technical-sessions/presentation/diegues
16. Diegues, N., Romano, P., Garbatov, S.: Seer: Probabilistic scheduling for hardware transactional memory. In: Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '15, pp. 224–233. ACM, New York, NY, USA (2015). DOI 10.1145/2755573.2755578. URL http://doi.acm.org.focus.lib.kth.se/10.1145/2755573.2755578
17. Diegues, N., Romano, P., Rodrigues, L.: Virtues and limitations of commodity hardware transactional memory. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14, pp. 3–14. ACM, New York, NY, USA (2014). DOI 10.1145/2628071.2628080. URL http://doi.acm.org/10.1145/2628071.2628080
18. Evan Jones: tpccbench. https://github.com/evanj/tpccbench (2007). Accessed 23 September 2019
19. Felber, P., Issa, S., Matveev, A., Romano, P.: Hardware read-write lock elision. In: Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, pp. 34:1–34:15. ACM, New York, NY, USA (2016). DOI 10.1145/2901318.2901346. URL http://doi.acm.org/10.1145/2901318.2901346
20. Garivier, A., Moulines, E.: On upper-confidence bound policies for switching bandit problems. In: Proceedings of

the 22Nd International Conference on Algorithmic Learning Theory, ALT'11, pp. 174–188. Springer-Verlag, Berlin, Heidelberg (2011). URL http://dl.acm.org/citation.cfm?id=2050345.2050365
21. Goel, B., Titos-Gil, R., Negi, A., McKee, S.A., Stenstrom, P.: Performance and energy analysis of the restricted transactional memory implementation on haswell. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 615–624 (2014). DOI 10.1109/IPDPS.2014.70
22. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, pp. 175–184. ACM, New York, NY, USA (2008). DOI 10.1145/1345206.1345233. URL http://doi.acm.org/10.1145/1345206.1345233
23. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proceedings of the 15th International Conference on Distributed Computing, DISC '01, pp. 300–314. Springer-Verlag, London, UK, UK (2001). URL http://dl.acm.org/citation.cfm?id=645958.676105
24. Hart, T.E., McKenney, P.E., Brown, A.D., Walpole, J.: Performance of memory reclamation for lockless synchronization. J. Parallel Distrib. Comput. **67**(12), 1270–1285 (2007). DOI 10.1016/j.jpdc.2007.04.010. URL http://dx.doi.org/10.1016/j.jpdc.2007.04.010
25. Issa, S.: P8TM source code and benchmarks. https://github.com/shadyalaa/POWER8TM (2017). Accessed 23 September 2019
26. Jacobi, C., Slegel, T., Greiner, D.: Transactional memory architecture and implementation for IBM System Z. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45, pp. 25–36. IEEE Computer Society, Washington, DC, USA (2012). DOI 10.1109/MICRO.2012.12. URL http://dx.doi.org/10.1109/MICRO.2012.12
27. Java docs: ReentrantReadWriteLock. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html (2018). Accessed 23 September 2019
28. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid transactional memory. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, pp. 209–220. ACM, New York, NY, USA (2006). DOI 10.1145/1122971.1123003. URL http://doi.acm.org/10.1145/1122971.1123003
29. Lai, T., Robbins, H.: Asymptotically efficient adaptive allocation rules. Advances in Applied Mathematics (1985)
30. Le, H.Q., Guthrie, G.L., Williams, D.E., Michael, M.M., Frey, B.G., Starke, W.J., May, C., Odaira, R., Nakaike, T.: Transactional memory support in the IBM POWER8 processor. IBM Journal of Research and Development **59**(1), 8:1–8:14 (2015). DOI 10.1147/JRD.2014.2380199
31. Marathe, V.J., Scherer, W.N., Scott, M.L.: Adaptive software transactional memory. In: Proceedings of the 19th International Conference on Distributed Computing, DISC'05, pp. 354–368. Springer-Verlag, Berlin, Heidelberg (2005). DOI 10.1007/11561927_26. URL http://dx.doi.org/10.1007/11561927_26
32. Matveev, A., Shavit, N.: Reduced Hardware NOrec: A safe and scalable hybrid transactional memory. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, pp. 59–71. ACM, New York, NY, USA (2015). DOI 10.1145/2694344.2694393. URL http://doi.acm.org/10.1145/2694344.2694393
33. Mckenney, P.E., Slingwine, J.D.: Read-Copy Update: Using Execution History to Solve Concurrency Problems. In: Parallel and Distributed Computing and Systems, pp. 509–518. Las Vegas, NV (1998)
34. McKenney, P.E., Walpole, J.: What is RCU, Fundamentally? https://lwn.net/Articles/262464/ (2007). Accessed 23 September 2019
35. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on, pp. 35–46. IEEE (2008)
36. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In: Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ISCA '15, pp. 144–157. ACM, New York, NY, USA (2015). DOI 10.1145/2749469.2750403. URL http://doi.acm.org/10.1145/2749469.2750403
37. Nussbaum, D., Lev, Y., Moir, M.: PhTM: Phased transactional memory. In: The Second ACM SIGPLAN Workshop on Transactional Computing, TRANSACT '07. ACM, New York, NY, USA (2007). URL https://urresearch.rochester.edu/institutionalPublicationPublicView.action?institutionalItemId=4058
38. Ortner, R., Ryabko, D., Auer, P., Munos, R.: Regret bounds for restless markov bandits. In: Proceedings of the 23rd International Conference on Algorithmic Learning Theory, ALT'12, pp. 214–228. Springer-Verlag, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-34106-9_19. URL http://dx.doi.org/10.1007/978-3-642-34106-9_19
39. Raminhas, P., **Issa**, t., Romano, P.: Enhancing efficiency of hybrid transactional memory via dynamic data partitioning schemes. In: Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid '18, pp. 1–11 (2018). DOI 10.1109/CCGRID.2018.94
40. Rossbach, C.J., Hofmann, O.S., Witchel, E.: Is transactional programming actually easier? In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, pp. 47–56. ACM, New York, NY, USA (2010). DOI 10.1145/1693453.1693462. URL http://doi.acm.org/10.1145/1693453.1693462
41. di Sanzo, P., Quaglia, F., Palmieri, R.: Analytical modelling of commit-time-locking algorithms for software transactional memories. In: Int. CMG Conference (2010)
42. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, pp. 1150–1160. VLDB Endowment (2007). URL http://dl.acm.org/citation.cfm?id=1325851.1325981
43. TPC Council: TPC-C Benchmark. http://www.tpc.org/tpcc (2011). Accessed 23 September 2019
44. Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R., Michael, M.: Evaluation of Blue Gene/Q hardware support for transactional memories. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, pp. 127–136. ACM, New York, NY, USA (2012). DOI 10.1145/2370816.2370836. URL http://doi.acm.org/10.1145/2370816.2370836

45. Wang, Q., Kulkarni, S., Cavazos, J., Spear, M.: A transactional memory with automatic performance tuning. ACM Trans. Archit. Code Optim. **8**(4), 54:1–54:23 (2012). DOI 10.1145/2086696.2086733. URL `http://doi.acm.org/10.1145/2086696.2086733`
46. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pp. 19:1–19:11. ACM, New York, NY, USA (2013). DOI 10.1145/2503210.2503232. URL `http://doi.acm.org/10.1145/2503210.2503232`
47. Yu, J.Y., Mannor, S.: Piecewise-stationary bandit problems with side observations. In: Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09, pp. 1177–1184. ACM, New York, NY, USA (2009). DOI 10.1145/1553374.1553524. URL `http://doi.acm.org/10.1145/1553374.1553524`